UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

OBIWAN: Middleware for Memory Management of Replicated Objects in Distributed and Mobile Computing

Luís Manuel Antunes Veiga
(Mestre)

Dissertação para obtenção do Grau de Doutor em
Engenharia Informática e de Computadores

Outubro 2006

.

Dissertação realizada sob a orientação do

Prof. Paulo Jorge Pires Ferreira

Professor Associado do Departamento de Engenharia Informática

do Instituto Superior Técnico da Universidade Técnica de Lisboa

Publications

The work and results presented in this dissertation were partially supported, in chronological order, by Microsoft Research and by the Portuguese Science and Technology Foundation (*Fundação para a Ciência e Tecnologia, Ministério da Ciência e do Ensino Superior*). They are partially described in the following peer-reviewed scientific publications:

International Journals

1. Complete Distributed Garbage Collection, An Experience With ROTOR. (Selected papers from Rotor Workshop) Luís Veiga, Paulo Ferreira. In **IEE Research Journals - Software**, vol. 150(5), October 2003.

2. OBIWAN - Design and Implementation of a Middleware Platform. Paulo Ferreira, Luís Veiga, Carlos Ribeiro. In **IEEE Transactions on Parallel and Distributed Systems**, vol. 14(11), November 2003.

Book Chapters

3. Mobile Middleware - Seamless Service Access via Resource Replication. Paulo Ferreira, Luís Veiga. In **The Handbook of Mobile Middleware, Paolo Bellavista and Antonio Corradi, Auerbach Publications, Taylor and Francis-CRC Press**, October 2006.

International Conferences and Workshops

4. Mobility and Wireless Support in OBIWAN. Luís Veiga, Paulo Ferreira. In Advanced Topic Workshop on Middleware for Mobile Computing, **3rd IFIP/ACM Middleware Conference**, Heidelberg, Germany, November 16, 2001.

5. Incremental Replication for Mobility Support in OBIWAN. Luís Veiga, Paulo Ferreira. In **22nd IEEE International Conference on Distributed Computing Systems (ICDCS'02)**, Vienna, Austria, July 2-5, 2002.

6. Mobility Support in OBIWAN. Luís Veiga, Paulo Ferreira. In **2nd Microsoft Research Summer Workshop**, Cambridge, UK September 9-11, 2002.

7. RepWeb: Replicated Web with Referential Integrity. Luís Veiga, Paulo Ferreira. In **18th ACM Symposium on Applied Computing (SAC'03)**, Melbourne, Florida, USA, March 9-12, 2003.

8. Complete Distributed Garbage Collection, An Experience With ROTOR. Luís Veiga, Paulo Ferreira. In **2nd Microsoft Research Annual SSCLI Workshop**, Redmond, WA, USA, September 17-19, 2003.

9. Turning the Web into an Effective Knowledge Repository. Luís Veiga, Paulo Ferreira. In **6th International Conference on Enterprise Information Systems**, Porto, Portugal, April 14-17, 2004.

10. Transaction Policies for Mobile Networks. Nuno Santos, Luís Veiga, Paulo Ferreira. In **5th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy 2004)**, New York, NY, USA, June 7-9, 2004.

11. Loosely-Coupled, Mobile Replication of Objects with Transactions. Luís Veiga, Nuno Santos, Ricardo Lebre, Paulo Ferreira. In Workshop on Qos And Dynamic Systems. **10th IEEE International Conference on Parallel And Distributed Systems (ICPADS)**, New Port Beach, CA, USA, July 5-9, 2004.

12. PoliPer: Policies for Mobile and Pervasive Environments. Luís Veiga, Paulo Ferreira. In 3rd Workshop on Adaptive and Reflective Middleware, **6th ACM/IFIP/USENIX Middleware**, Toronto, Ontario, Canada, October 18-22, 2004.

13. Asynchronous Complete Distributed Garbage Collection. Luís Veiga, Paulo Ferreira. In **19th IEEE international Parallel and Distributed Processing Symposium (IPDPS)**, Denver, CO, USA, April 4-8, 2005.

14. Extending .Net Remoting with Distributed Garbage Collection. Paulo Pereira, Luís Veiga, Paulo Ferreira. In **2nd International Conference on Innovative Views of .Net Technologies (IV.Net 2006)**, Brazil, October 21, 2006.

Other Publications

15. Garbage Collection Curriculum. Paulo Ferreira, Luís Veiga. In **MSDN Academic Alliance Curriculum Repository**, Object ID 6812, powerpoint document containing 144 slides, available at http://www.msdnaacr.net/curriculum/pfv.aspx?ID=6182, Microsoft, July 2005.

Resumo

Actualmente, o desenvolvimento de aplicações para ambientes móveis é complexo pois os programadores têm de lidar com várias questões de nível-sistema (e.g., funcionamento em modo desligado, gestão de memória, variabilidade no ambiente). Isto impede-os de se cingirem à lógica aplicacional, com consequentes perdas de produtividade e robustez das aplicações.

Nesta dissertação propomos uma plataforma *middleware* (OBIWAN) que permite melhorar a produtividade dos programadores e a robustez das aplicações, abordando três aspectos: suporte à execução de aplicações distribuídas através da replicação incremental e transparente de objectos, reciclagem distribuída de memória, e suporte ao nível-sistema para a adaptabilidade de aplicações.

A replicação proposta permite às aplicações, em sistemas distribuídos e móveis, continuarem a operar durante períodos de desconexão ou reduzida largura de banda, com melhor desempenho quando comparado com acesso remoto, sem necessidade de modificar as máquinas-virtuais em uso.

A reciclagem automática de memória (GC) liberta os programadores da gestão manual de memória. Propomos a primeira solução completa de GC distribuído para objectos replicados. Os algoritmos propostos são escaláveis, não interferem com as aplicações, e são facilmente integráveis nos sistemas actuais.

Para permitir a configuração do ambiente de execução a diferentes dispositivos e recursos nestes sistemas, a arquitectura proposta suporta a adaptabilidade e flexibilidade exigidas em ambientes móveis.

Palavras chave:

sistemas distribuídos, mobilidade, *middleware*, replicação, reciclagem automática de memória distribuída, adaptabilidade.

Abstract

Developing applications for mobile environments is difficult because the programmers are currently forced to deal with various system-level issues (e.g., operation in disconnected mode, memory management, variability of surrounding environment). This prevents them from focusing on application logic, with consequent lower productivity and reduced application robustness.

We propose a middleware platform (OBIWAN) that helps programmers' productivity and application robustness, addressing three issues: support for execution of distributed applications by means of transparent incremental replication, distributed garbage collection, and system-level support for application adaptability.

Replication allows applications to keep on working while disconnected or with low-bandwidth access, with improved performance when compared with remote access. We propose incremental replication of object graphs, handling object-faults in distributed and mobile systems, without requiring modifications to underlying virtual machines.

Garbage collection (GC) releases programmers from manual memory management. We propose the first complete distributed GC solution for replicated objects. The algorithms proposed are scalable, non-disruptive to applications, and are easily integrated with current off-the-shelf systems.

A natural follow-up of these subjects is to address the automatic adaptation and configuration of the execution environment, to different devices and available resources in these systems. The architecture proposed provides the necessary entry-points, supporting the flexibility and adaptability required by mobile applications.

Keywords:

distributed systems, mobility, middleware, replication, distributed garbage collection, adaptability.

Agradecimentos

Gostaria de agradecer enfaticamente ao meu orientador científico, Prof. Paulo Ferreira, por acreditar no meu trabalho e pelos frequentes incentivos e palavras de encorajamento, quer para o decurso do trabalho desta dissertação, quer para o restante da minha actividade científica e docente. O parágrafo anterior aplicava-se em 2001, aquando da realização do mestrado, e mantém-se de igual forma hoje.

Aos membros da Comissão de Acompanhamento de Tese: Prof. José Legatheaux Martins, Prof. José Alves Marques, Prof. Arlindo Oliveira e Prof. Paulo Ferreira.

Ao Departamento de Engenharia Informática do IST, pelo usufruto de 29 meses de Dispensa de Serviço Docente, nas pessoas do seu actual Presidente, Prof. José Alves Marques, e dos Coordenadores do Programa de Doutoramento em Engenharia Informática, em cujo consulado este trabalho teve lugar: Prof. Joaquim Jorge e Prof. Alberto Silva.

À Fundação para a Ciência e Tecnologia, à Microsoft Research, e à Fundação Luso-Americana para o Desenvolvimento, pelo apoio financeiro prestado.

A todos os membros, em geral, do Grupo de Sistemas Distribuídos do INESC-ID em Lisboa, presentes e passados.

Ao Eng. Nuno Santos pelo frutuoso trabalho em equipa tanto na investigação como na docência.

Aos membros do GSD com quem tenho colaborado no âmbito de artigos científicos, projectos de investigação e discussão de ideias: Prof. Carlos Ribeiro, Prof. Rodrigo Rodrigues, Eng. João Garcia, Eng. João Silva, Eng. João Barreto, Eng. Ricardo Lebre.

Aos alunos que tive oportunidade de co-orientar na realização do seu Trabalho Final de Curso, pelo seu entusiasmo e empenho: Engs. Nuno Santos, Ricardo Mendes, Ezequiel Alabaça, Gustavo Guerra, Miguel Cartó, João Salavessa, José Cunca, Paulo Gonçalves, José Lopes, Tiago Bernardo, e Edgar Marques.

Ao Prof. David Matos, pela graciosa disponibilização do *template* utilizado na redacção desta dissertação.

Aos meus amigos por sempre o terem sido. Àqueles que, embora não conhecendo pessoalmente, foram o apoio preci(o)so nas horas de trabalho mais solitário, aos músicos.

A todos os membros, presentes e passados, do colectivo progressivo Solarys.

A meus pais, Manuel Joaquim e Ana Maria, e irmãos, Carlos e Ana.

A Andreia, pelo apoio, carinho, compreensão, e pelas horas de companhia perdidas.

Luís Manuel Antunes Veiga
Parque das Nações, 23 de Outubro de 2006

Contents

List of Figures

List of Tables

I

Global Overview

(this page was intentionally left blank)

I.1 Introduction

A beginning is a very delicate time... – in "Dune", Frank Herbert

...historical perspective...

Personal computing emerged during the 1980s. Desktop machines, with reasonable computing power, have only become common, in the homes of developed countries, during the latter half of the decade. In institutional and corporate organizations, though, their spreading had begun earlier. The 1990s witnessed the global dissemination of networking, and with it, of classic distributed computing. First, pursuing the trend (that initiated still in the earlier decade) of connecting computers, in local networks of offices and organizations. Even though the Internet with e-mail services and various file transfer infrastructures (Bhushan 1971) had existed for a long time at universities, it was nonetheless only later during the decade, that wide area distributed systems became a commodity, with the advent of the World Wide Web (Berners-Lee 1994; Berners-Lee et al. 1994).

A similar tendency has been observed in the fields of mobile and pervasive computing. Although mobile computing has been a field of active research for over a decade (Satyanarayanan 1996), it was only in recent years that mobile access to computer networks has manifolded, both in number of users and variety of applications, as far as the general public is concerned. Pervasive computing, as such, has been postulated as early as 1988 and described in (Weiser 1991; Weiser 1993), championing for *disappearing (or invisible) computing*, where a myriad of tiny devices would seamlessly cooperate. Pervasive computing can be regarded as a possible evolution of mobile computing, which has only recently became prevalent with ad-hoc networks, and standards for short-range wireless communication such as Bluetooth and RFID.[1] In this sense, mobile computing promises computing *anywhere, anytime*. Pervasive computing, on its turn, could be described as computing *all-the-time, everywhere*. While the fields of distributed, mobile and pervasive computing share common aspects, the work presented in this dissertation is focused primarily on the areas of distributed and mobile computing.

I.1.1 Motivation, Goals and Challenges

...the case for mobile computing...

The fundamental paradigm shift from classic distributed computing to mobile computing lies in the notion that a user should no longer be forced, in order to operate with a computer or

[1] Radio Frequency IDentification.

a network, to be physically close to a stationary machine that is connected to the fixed network. Additionally, even if there are many such fixed access machines available, the user due to its movements may still choose to transfer his/her data among different machines.

We witness presently, and for some time now, to a vast dissemination of portable devices (e.g., laptops, tablet PC's, PDAs) made available by various manufacturers. This makes mobile and pervasive computing a reality. However, compared to classic distributed computing, mobile computing raises additional difficulties:

- variable node topology (that is assumed to be static in the classic distributed scenario);
- slower, possibly unreliable connections, frequent disconnections (including voluntary ones to reduce cost);
- mobile devices (e.g., PDAs) often have limited resources such as processing power, memory, and storage capabilities (e.g., PDAs have no disks);
- limited battery lifetime.

These difficulties result from the fact that mobile networks are characterized by the mobility and intermittent activity (due to shut down or disconnection) of one or more hosts. Nonetheless, it is well known that a large number of applications in these environments need to share information. However, programmers do have a real hard task while developing mobile distributed applications in which sharing is needed. Furthermore, programmers are forced to deal with system level issues such as data replication, memory management, consistency, durability, availability, security, etc.

In summary, middleware support for mobile computing should offer the following functionality and properties:

1. To leverage mobile devices computing-power, and its resources in general. In fact, recent evolution in mobile devices power and resources makes them viable platforms to perform a large number of tasks (besides just simple interface devices);
2. To minimize and/or prevent dependency on continuous connection to servers. Applications should be able to perform useful work even when disconnected from the network;
3. To allow efficient, productive and broad-based application development for these environments. In other words, to avoid the obligation to use a strict file, or query-based kind of application. Instead, a rich object-orientation approach should be provided.

Thus, the overall goal of this work is to facilitate application development for mobile computing and wide-area networks, i.e., to ease programmers' lives. We propose an approach based on an object-oriented middleware platform to achieve this. The main advantage that stems from using middleware is obviating the need to modify the underlying operating system and virtual machine. Object-oriented programming is arguably the most widespread paradigm for application development.

To achieve the aforementioned goal there are still some challenges to address:

- Frequent disconnection - this is addressed with data replication.
- Programming reliability (referential integrity) and memory limitations - this is addressed with automatic memory management.
- Variability in the environment - this is addressed with support for policies and mechanisms to provide adaptability to applications.

More specifically, within each one of these challenges, there are a number of problems to address. We present them in detail in Section I.1.3.

I.1.2 Proposed Approach

In this section we present the main aspects of our proposed approach to address the goals and challenges mentioned above. W.r.t. each aspect, we present a case for its importance, offering a number of arguments justifying its relevance. We address: i) support for replication, ii) use of object-oriented programming, iii) deployment via middleware, and iv) non-intrusiveness of the middleware.

...the case for replication...

Traditional distributed applications are client-server based. These can either be always connected, i.e. need a permanent connected channel or be occasionally connected. In the context of mobile computing, an always-connected approach is not an adequate solution for economic and technical reasons, and w.r.t. performance. Economically, users may not wish to be online all the time due to communication costs. Technically, signal power may be insufficient, in some areas, to maintain constant connectivity and a strongly-coupled approach limits flexibility and scalability. Moreover, being always online imposes important energy drains, which limits autonomy. W.r.t. performance, applications will be able to execute faster if data and code are collocated, instead of on separate machines.

With such a client-server solution, availability is strictly dependent on connectivity, i.e., connection breakdown forces applications to stop. In the case of limited areas where wireless connectivity is almost permanent, and deemed inexpensive by users, network bandwidth may still be significantly lower than on a fixed network. Even if there is high bandwidth available, local access to data will still be faster than accessing remote data.

On the other hand, in most cases, mobile devices are still regarded as small color-screen, sound-enabled clients mainly used for interface purposes. This is the case with common mobile database-oriented applications where data queries are performed, ultimately, on the servers. Clients running on devices simply ask for user input and present query results in a more or less structured and visually appealing manner. This increases load on the servers and dependency on connectivity.

Data replication is a well-know technique for improving data availability and application performance as it allows to collocate data and code. Thus, data availability is ensured because, even if the network is not operational, data remains locally available; in addition, application performance is potentially better (when compared to a remote access/invocation approach) as all accesses to data are local. Thus, data replication as an enabling mechanism, is specially suited for mobile computing environments, in a variety of scenarios.

...the case for object-oriented programming...

In general, applications may be developed according to several different paradigms. These depend on the different abstractions supported by the programming language, corresponding application programming interface, and the underlying mobile middleware. For example, the middleware may simply provide a file system API, or it may support more complex abstractions such as tuples, relational entities, or objects.

One relevant issue regarding such paradigms is their ability, or lack of it, to support arbitrary graphs of data. In general, w.r.t. all data representation abstractions aforementioned, data graphs are the most flexible way to represent and manipulate (via navigation) structured application data, despite specific scenarios where others are either specially suited, or the only ones available for simplicity. The existence of data graphs has a strong impact when deciding which and how data must be replicated.

The object-oriented paradigm naturally supports such notion of data graphs in the form of object graphs, and navigation via object referencing and de-referencing. Obviously, the same applies to structured files whose contents include references to other files (e.g., graphs of HTML files connected by URLs, or XML documents storing pieces of data that reference elements in different XML documents). Whereas relational entities can refer to each other by means of foreign keys, therefore creating a graph, applications access such data by performing explicit queries. Thus, there is no navigation on such a graph as it happens with object-based applications.

Object-oriented programming is therefore the best suited approach to manipulate graph-based data because it concomitantly leverages :

- a well established foundation of programming languages,
- arguably the widest audience of practitioners/programmers already versed in it.
- a vast repository of existing code and applications that may thus be used in and adapted to mobile environments.

Thus, support for object oriented programming is an important requirement for widespread mobile application development.

...the case for middleware supporting object replication...

When left to the programmer, such system level issues like the ones mentioned earlier (data replication, memory management, consistency, durability, availability, security, etc.) are the cause of errors, low productivity and useless applications. Middleware layers, residing between the operating system (or, when present, virtual machines) and applications, release the programmer from these system-level issues. However, when using a middleware platform, programmers are often forced to use a particular programming paradigm which may not be the most suited to the particular application being developed. Currently, dealing which such paradigm diversity implies: i) either using different middleware platforms, with obvious inconveniences such as integration problems and learning costs, or ii) when restricted to one middleware platform the programmers are forced to deal with system-level issues such as handling the creation of replicas and the corresponding consequences in terms of object faulting, memory management, among other details.

A common example is that of situations in which an application would be preferably developed using mobile agents instead of traditional remote object invocation (RMI). There are also circumstances in which, instead of invoking an object remotely, it would be more adequate, in terms of performance and network usage, to create a replica of the object and invoke it locally.

Due to the present and foreseen ubiquity of mobile devices, programmers should be able to develop applications for mobile environments, making use of replication, leveraging the most of their knowledge (i.e., object-orientation paradigm, programming languages, development environments) while retaining the flexibility to address (not so) unexpected, while frequent, situations correctly. Thus, in the context of this dissertation, we are interested mainly in the development of middleware to correctly support data replication (as a mechanism) to object-oriented programming (as a paradigm) .

This methodology, object-oriented programming enhanced with object replication, may be applied to the development of applications of several kinds. Possible examples include applications with geographically dispersed data sources, with replicated data and mobile users. This is the case in the fields of cooperative work for engineering projects, design, etc., performed by virtual organizations. There are also entertainment applications, such as games, that manipulate large volumes of structured data, where mobile users (players) only load, or have access to, a fraction of the whole universe, during a given time. Applications resorting to simulated environments, embodying the notions of navigation, spatial placement, and hierarchy of scene elements, naturally lend themselves to represent data as a replicated object graph. Mobile agents that need to replicate data between hosts can also profit from this approach. Other examples include collaborative *mind-mapping* tools (Mueller et al. 2003) that allow several users to visualize and manipulate object graphs representing concepts, ideas, or tasks.

The notion of replicated graph is also present in the context of collaborative applications for content authoring and management. In these applications, regardless of their thematic, data is ultimately represented via structured (connected) elements and files, coded in HMTL and/or XML (e.g., web sites, distributed XML repositories (Abiteboul et al. 2003), Wikis (Cunningham and Leuf 2001; Curran et al. 2004)).

...the case for non-intrusive middleware...

Finally, portability and programmability are also relevant aspects that must be taken into account by the mobile middleware. As a matter of fact, mobile environments are characterized by the heterogeneity of devices, operating systems, virtual machines, etc. So, the mobile middleware should be, as much as possible, independent from such differences in order to be portable to a wide range of platforms.

Programmability means that the middleware should release application programmers from dealing with system level issues while providing an API familiar to programmers used to develop distributed applications in general. Thus, the mobile middleware should not imply the modification of neither the operating systems nor the virtual machines, and should not impose radically new APIs.

I.1.3 Problems to Address

There are several relevant difficulties that must be solved to take full advantage of replication, in this case, in the context of object-oriented programming for mobile environments. In this dissertation we address the following:

Replica Management: Replica management is related to the fundamental issues of knowing which and how data should be replicated. The fact that mobile devices impose severe constraints in terms of memory and communication resources raises the importance that the data to be replicated should be the one that is really needed, so that memory is not wasted. Replication should also be performed while attempting to mitigate network limitations (e.g., latency).

Memory Management: Memory management is related to two issues: i) to preserve referential integrity of the object graph (by freeing the developer from error-prone explicit memory management), and ii) to ensure that the memory of computers (including mobile devices such as PDAs, laptops, etc.) is occupied with useful data. This implies freeing the memory occupied by useless replicas. Thus, replicated data that is no longer needed must be automatically detected and reclaimed, thus releasing the memory occupied, which can be done by garbage collecting such replicas.

Adaptability: Adaptability is the capability that applications have to control and adapt to the available resources (memory, network, etc.) in order to better deal with the variability of mobile environments; such variability affects network bandwidth, network connection or disconnection, amount of available memory, etc. Therefore, it is advantageous that the underlying middleware supports flexible mechanisms to enable applications to react and adapt. This way, they can withstand the dynamics of mobile environments (e.g., variable network availability, amount of free memory on the device). In particular, regarding memory limitations in mobile devices, mechanisms should be provided that allow swapping-out useful data to remote computers.

The above mentioned issues are more and more relevant as we move from a traditional wired network of desktop computers to an environment formed by mobile devices able to wirelessly connect to the fixed network or take part in ad-hoc networks. As a matter of fact, mobile devices, when compared to desktop computers, are much more resource-constrained in terms of memory, network availability and bandwidth, battery, etc.; in addition, applications running on such devices face a much more dynamic environment given the natural movement of users and devices.

Note that many other issues are equally important (Satyanarayanan 1996), such as security, consistency, power management, fault-tolerance, but these are out of the scope of this dissertation.

There are a number of systems that address one or more of the above mentioned problems. However, none of them addresses all the problems in an integrated manner, while providing a portable approach. There are systems that provide support for replication but lack either transparency w.r.t. traditional object-oriented programming, or enforcement of referential-integrity, or adaptability w.r.t the number of replicated objects each time the network is accessed. There are systems that provide distributed garbage collection but lack either safety or completeness in the presence of replicated objects. Finally, w.r.t. adaptability, most systems lack support for object-swapping and the ones that do support it, achieve this using approaches that require changes to existing virtual machines.

I.1.4 Contributions

The work presented in this dissertation was developed mainly in the context of OBIWAN (**O***bject* **B***roker* **I***nfrastructure for* **W***ide* **A***rea* **N***etworks*), a middleware platform designed by the author to support the development of applications using object-oriented languages, enabled with object replication, adequate memory management, and adaptability. OBIWAN also provides support for mobile agents.

This dissertation presents contributions in the following areas:

Support for Object Replication: Novel support for object replication in mobile environments without imposing changes to the underlying virtual machine (Veiga and Ferreira 2002a; Veiga and Ferreira 2002b; Ferreira et al. 2003; Veiga et al. 2004). Incremental object replication and dynamic object clustering provide the necessary flexibility to deal with mobile environments.

- Transparent and flexible handling of object-faults in distributed systems.
- Incremental replication of graphs of objects.
- Dynamic clustering of replicated objects.
- Support for external data consistency mechanisms (e.g., optimistic transactions) applied to replicated objects, as employed in (Santos et al. 2004).

Distributed Garbage Collection: Scalable, asynchronous and complete garbage collection for distributed systems with and without replication. Two novel algorithms for distributed garbage collection and the first viable solution to achieve complete distributed garbage collection for replicated object systems.

- Distributed garbage collection algorithms, for distributed objects in wide area systems, able to detect and reclaim distributed cycles of garbage. The first algorithm uses a centralized detection approach (Veiga and Ferreira 2003a; Veiga and Ferreira 2003b), also employed in (Pereira et al. 2006). The second algorithm is able to perform de-centralized detection (Veiga and Ferreira 2005a).
- Distributed garbage collection algorithm, for wide area systems, able to detect and reclaim distributed cycles of garbage comprising replicated objects (Veiga and Ferreira 2005b).

Adaptable Object Replication and Memory Management: A policy-based system applied to management of object replication and memory management, alongside with other mechanisms to handle memory shortage in mobile devices (Veiga and Ferreira 2004a; Veiga and Ferreira 2005c).

- Dynamic adaptation of the parameters of object replication mechanisms, such as the amount of objects to replicate at a given time, to account for variations of the execution environment in mobile networks with constrained devices (e.g., available memory, bandwidth, connection quality, neighboring devices, application counterparts running in different devices).
- Transparent *object-swapping*, an aggressive memory management mechanism that consists in the de-localization of objects reachable to applications. It addresses, and masks from programmers, and (as possible) from users, the memory limitations of constrained devices.

Additional implementation work regarding Object Replication:

- Implementation of OBIWAN for laptop and desktop machines, and for resource constrained devices in mobile networks.
- Integration of OBIWAN with i) a commercial web and application server; ii) persistent repositories (both relational and OO-based); and iii) a commercial application development environment (Visual Studio.Net 2005).

Additional implementation work regarding Distributed Garbage Collection:

- Implementation of Complete Distributed Garbage Collection algorithms for the Microsoft Shared Source Common Language Runtime (also known as *Rotor*), and for OBIWAN.
- Application of distributed garbage collection algorithms, able to detect distributed cycles of garbage, to enforce referential integrity and perform complete memory management in web systems (Veiga and Ferreira 2003c; Veiga and Ferreira 2004b) with dynamic content generation, both with and without replication.
- Distributed Garbage Collection Simulator implemented in Lisp.

I.1.5 Document Roadmap

The rest of this dissertation is organized as follows. It comprises five parts. The next chapter of Part I introduces an architectural overview of the OBIWAN object replication middleware platform (OBIWAN) and the environment it addresses. It introduces the main middleware components and data structures used throughout the rest of the document.

Part II describes the support for incremental object replication. Part III addresses the automatic memory management of distributed and replicated objects, i.e., distributed garbage collection algorithms. Part IV addresses adaptability of the middleware to changes in the environment. Part V concludes the dissertation presenting conclusions and introducing on-going and future work.

Parts II and III account for the most part of this dissertation. They share a common inner structure. They include individual chapters with: i) related work, ii) architecture/algorithms, iii) implementation, iv) evaluation, and v) conclusions.

I2 OBIWAN Architecture

This chapter presents the architecture of OBIWAN. It presents its rationale, usages and application development model. It describes its middleware components and main data structures, and how they are integrated. It is presented with increasing level-of-detail, in Figures I.2.1, I.2.2 and I.2.3, showing respectively: i) an overview of the OBIWAN application and usage model, ii) its network architecture and how data and functionality are spanned across nodes, and iii) the component structure of OBIWAN middleware running on particular each node. The architecture depicted here is common to the remaining parts of this dissertation. It also elucidates the aspects that will be detailed in the remainder of this document.

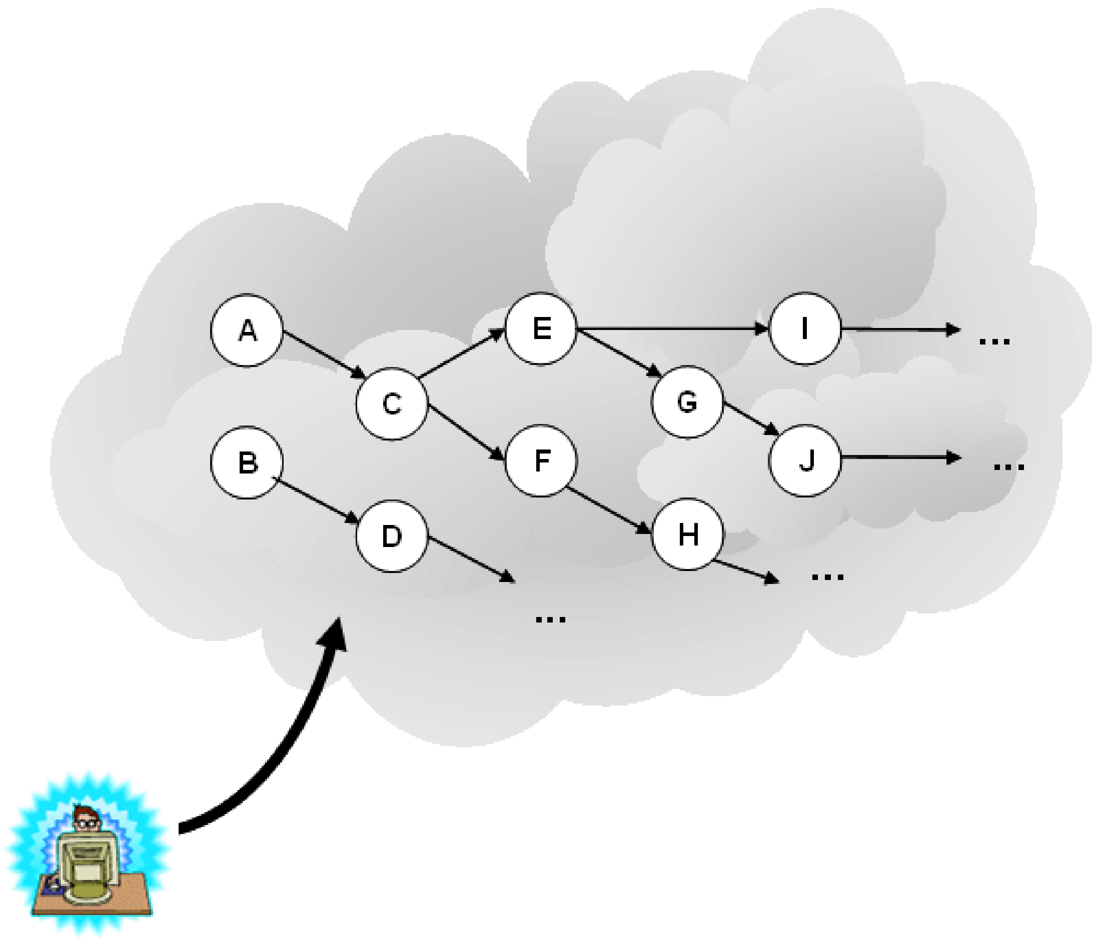


Figure I.2.1: OBIWAN model overview: applications manipulate oceans of objects.

OBIWAN addresses the general scenario in which a user will want to access data using a desktop PC in his office, using a laptop while in the airport or in the hotel, using a PDA in a taxi, etc. The user wants to live in this “data ubiquitous world” with no other concern besides doing his own work and, as much as possible, to keep on working in spite of any system problem that may occur (e.g., network partitions).

The overall objective of the OBIWAN project is to design and implement a system that: (i) is well suited to support mobile distributed applications with strong sharing needs in a mobile environment, and (ii) facilitates application development by releasing programmers from the need to handle complex system issues such as replication, memory management, etc., while providing the right level of abstraction and functionality to deal with unexpected situations.

OBIWAN is centered upon the notion of a generic object broker infrastructure to provide the means to support this sharing scenario. Intuitively, through this object broker infrastructure, OBIWAN supports applications that manipulate an *ocean of objects*. This is illustrated with the example portrayed in in Figure I.2.1.

Objects are arranged in the form of object graphs. Applications navigate in this ocean of objects by following references enclosed in objects. These graphs may be created and manipulated by means of object-oriented languages or composed of web documents.

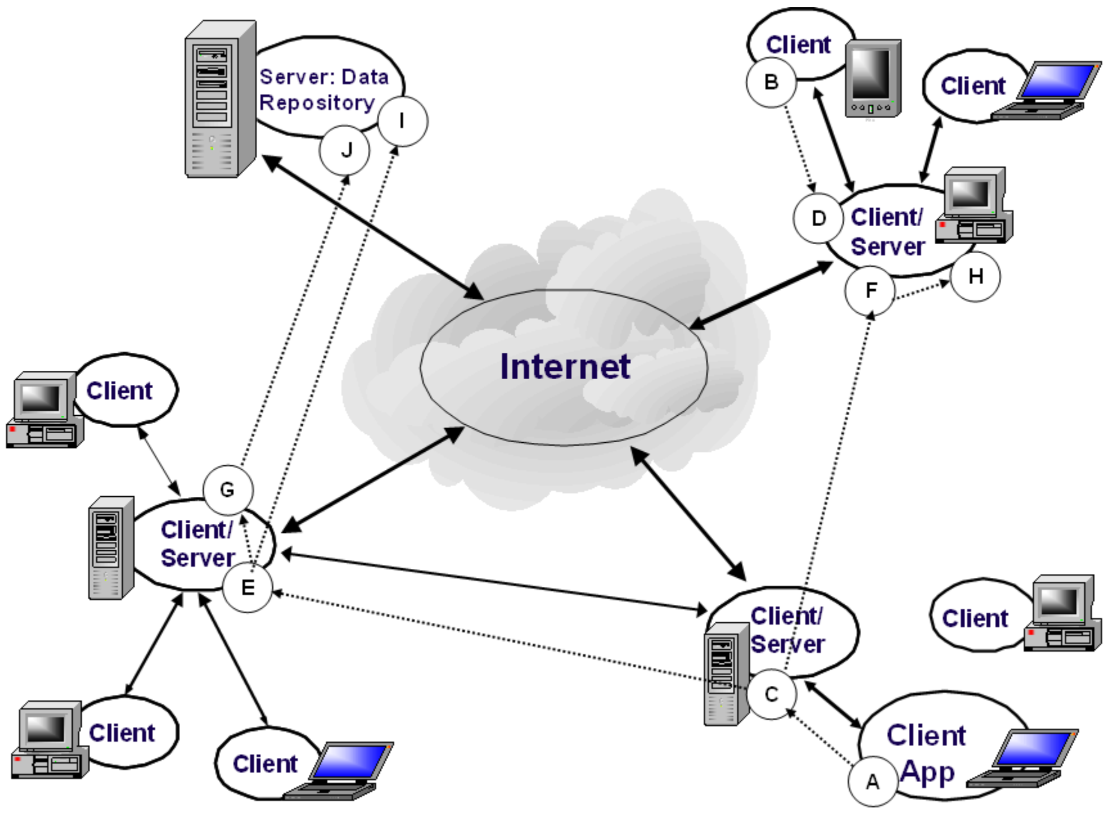


Figure I.2.2: OBIWAN Network Architecture

OBIWAN gives to the application programmer the view of a network of computers in which one or more processes run (see Figure I.2.2). Processes may hold objects (forming object graphs) and agents[1] (w.r.t. agents, we call such processes hosts). An object can be invoked locally (after being replicated) or remotely. Object graphs may thus span processes, as portrayed in Figure I.2.2. Mobile agents can be created and then freely migrated as long as some security policy allows. Objects may be scattered over a variety of locations and info-appliances, may be replicated among such appliances, and contain innumerous references connecting them.

Typically, applications invoke objects locally replicated into the computer where they are being executed. However, if desired they can explicit perform direct invocations on remote objects, without previously replicating them.

In general, every computer may hold objects and run user applications. In practice, the contribution of computers to object storage and application execution may be unbalanced. Extreme cases include servers that may act exclusively as object repositories, and mobile devices (possibly resource-constrained) that serve mainly to execute applications, thus replicating objects from other computers as needed by the users.

OBIWAN is a peer-to-peer middleware platform in the sense that any process may behave either as a client or as a server at any moment. In particular, w.r.t. replication this means that a process P can either request the local creation of replicas of remote objects (P acting as a client) or be asked by another process to provide objects to be replicated (P acting as a server).

I.2.1 OBIWAN Middleware Components

OBIWAN consists in a set of middleware component modules (see Figure I.2.3) which provide runtime services to applications and to other higher-level services. Runtime services execute on top of a virtual machine in every node. These are services that are available to the application regular code if explicitly wanted, but were designed to be used by code automatically generated that extends application code.

In the context of this dissertation, the most important modules are Object Replication and Memory Management (that are deployed on top of a set of core modules):

- **Object Replication:** This module provides the mechanisms for handling object-faults transparently, and supporting incremental object replication and dynamic clustering of replicated objects. This module is presented in Parts II and IV.
- **Memory Management:** This module is responsible for the distributed garbage collection, integration with the local garbage collector provided by the virtual machine, and *object-swapping*. Distributed garbage collection (both acyclic and cyclic) are described in Part III, while Part IV presents *Object-Swapping*. Memory management depends on object replication to be aware of which objects have been replicated in and out of the process.

The adaptability of these mechanisms is provided by a set of core modules that support the definition and enforcement of declarative policies:

[1]In OBIWAN, an agent is a composite of mobile code and the objects that comprise agent's state.

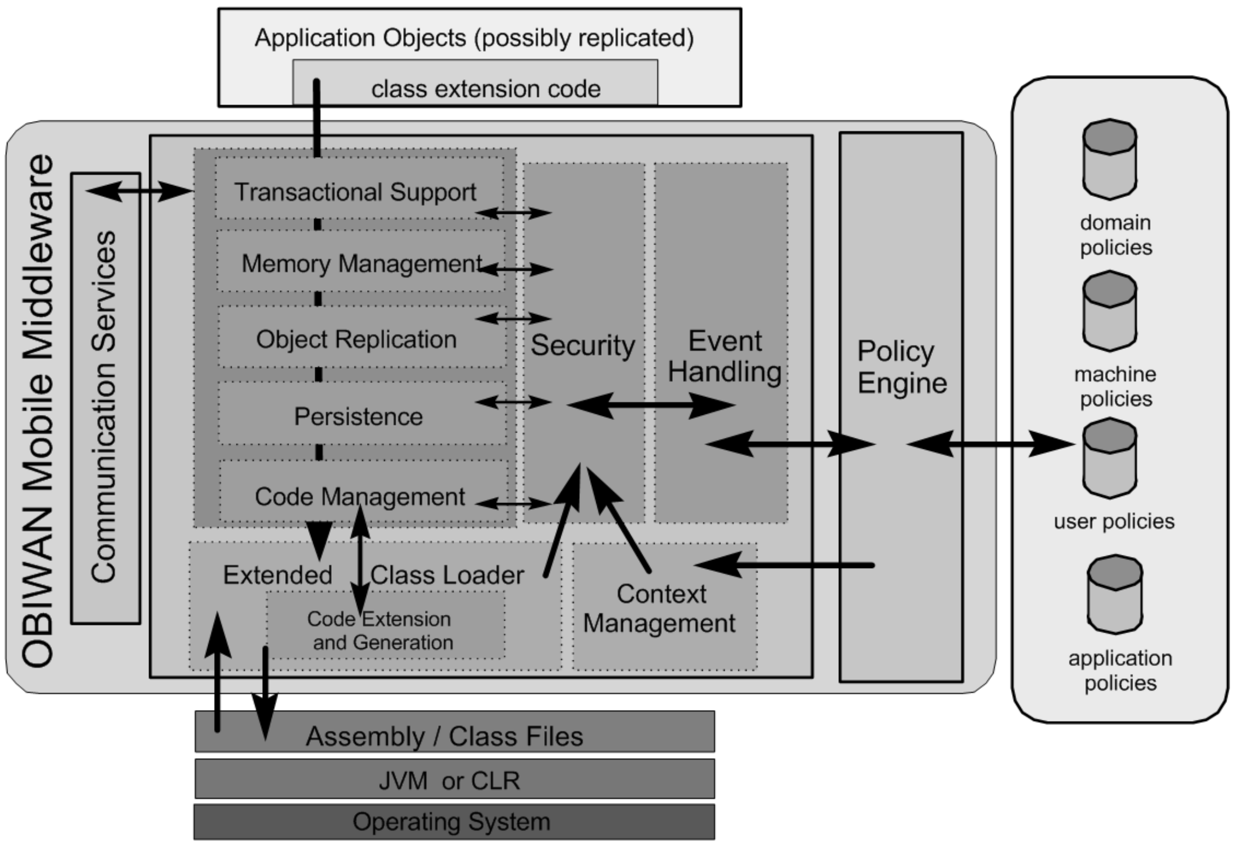


Figure I.2.3: OBIWAN Middleware Components.

- **Policy Engine:** The policy engine is the main inference component that manages, loads, and deploys declarative policies to oversee and mediate responses to events occurred in the system. It is described in detail in Part IV.

- **Event Handling:** In OBIWAN, notifications to applications and to the various system modules, are performed with resort to events. This module registers the relevant events, defined by policies, that may occur in the system (more details in Part IV);

- **Context Management:** This module abstracts resources and manages the corresponding properties whose values vary during applications execution (more details in Part IV).

Additionally, on top of the core modules, OBIWAN also considers the following modules that may be deployed via pre-defined policies:

- **Security:** The security module monitors all interactions between event-handling (trusted and linked to the policy engine) and every other module in the system. This module is out of the scope of this dissertation and will not be described in detail.

- **Extended Class Loader:** This module analyzes and extends application classes to make them replication-enabled, and generates code for proxies (Shapiro 1986). It makes use of an external application, *obicomp*, the OBIWAN compiler. This module is described in Part II, and also mentioned in Part III.

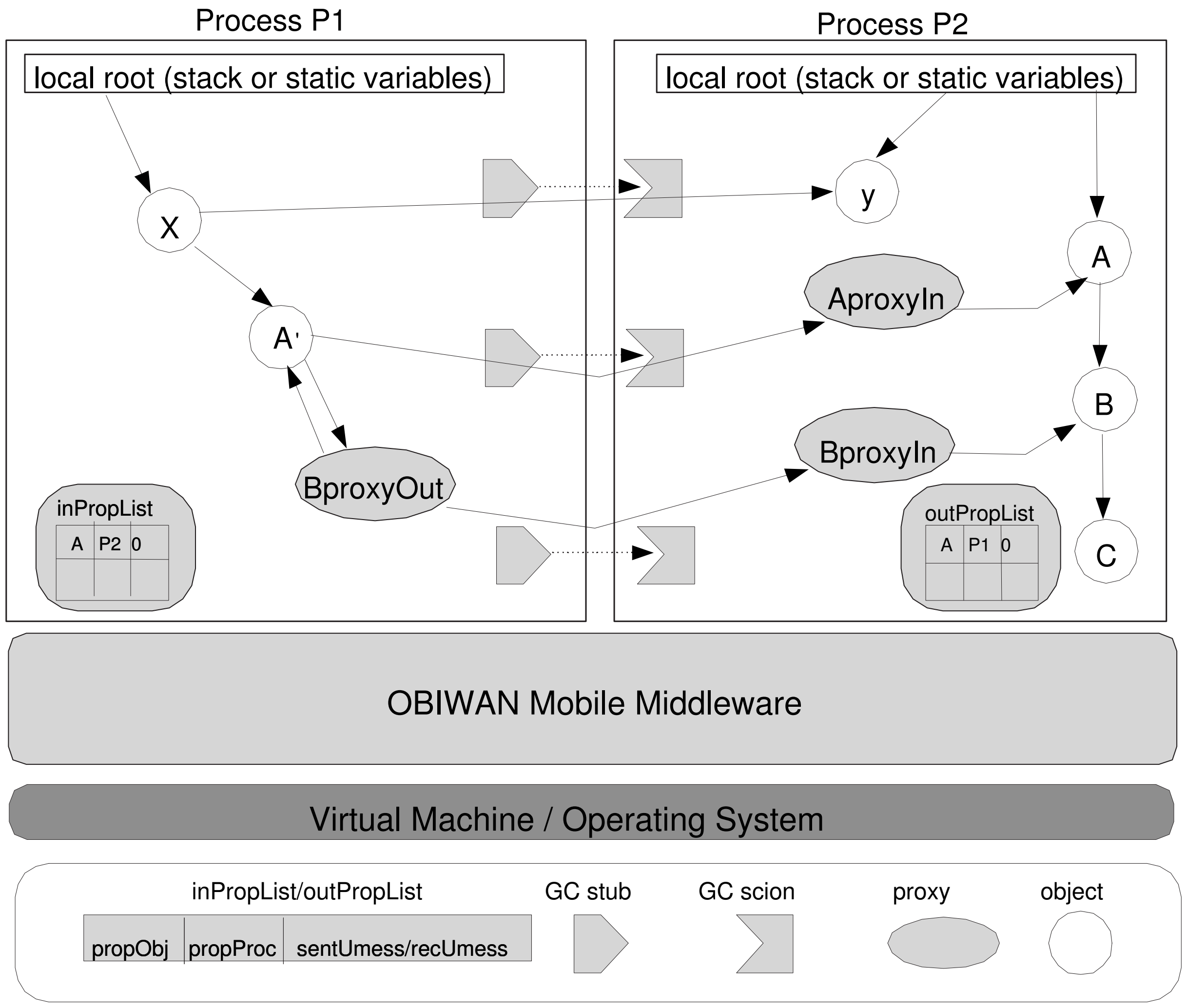


Figure I.2.4: Data structures supporting both object replication and memory management in OBIWAN. Light gray objects and data structures refer to OBIWAN middleware.

- **Code Management:** This module is an extension to the Extend Class Loader that allows the interception of class loading, during runtime, to detect and ensure that application code has been handled to run with OBIWAN. It is out of the scope of this dissertation.

- **Persistence:** This module handles persistent storage of object graphs. It makes use of an external tool and is described in Part II.

- **Transactional Support:** The transactional support module provides services to allow applications to manipulate objects within optimistic transactions. It is out of the scope of this document.

- **Communication Services:** This module abstracts all interactions with other nodes running OBIWAN, namely for object replication and exchanging DGC messages. It makes use of remote invocation mechanisms provided by the underlying virtual machines.

I.2.2 Data Structures

The fundamental data structures of the OBIWAN middleware, relevant to the remainder of this dissertation, are depicted in Figure I.2.4. It portrays a prototypical situation where process P1 is accessing objects that are being replicated from process P2. The middleware data structures defined and maintained by the Object Replication and Memory Management modules are:

Object Invocation:

- **Objects**: Applications invoke mostly local objects (such as X and Y), and replicated objects (e.g., A and A'). Replicated objects are those whose classes have been extended (or enhanced) to be handled by OBIWAN.
- **Proxies**: Proxies mediate remote execution and help in supporting object replication. Proxies may be proxy-out or proxy-in. A proxy-out stands in for an object that is not yet locally replicated (e.g., BProxyOut stands for B′ in P1). For each proxy-out there is a corresponding proxy-in.

Replication management:

- **InPropLists:** Entries in this list indicate which objects have been replicated *from* another process.
- **OutPropLists:** Entries in this list indicate which objects have been replicated *to* another process.

  Each entry of the inPropList/outPropList contains the following information: **propObj** is the reference of the object that has been replicated into/to a process; **propProc** is the process from/to which the object `propObj` has been replicated.

  Additionally, associated with each entry, there is a **sentUmess**/**recUmess** bit indicating if a unreachable message (for DGC purposes) has been sent/received (more details in Part III).

Memory management:

- **Stubs:** A GC-stub describes an outgoing inter-process reference, from a source process to a target process (e.g., from object X in P1 to object Y in P2).
- **Scions:** A GC-scion describes an incoming inter-process reference, from a source process to a target process (e.g., to object Y in P2 from object X in P1).

  GC-stubs and GC-scions do not impose any indirection on the native reference mechanism, i.e., they do not interfere either with the structure of references or the invocation mechanism. They are simply GC specific auxiliary data structures.

  Thus, GC-stubs and GC-scions should not be confused with (virtual machine) native stubs and scions (or skeletons) used for remote method invocations (RMI), that are used by proxies.

Through the management of these data structures, the middleware is able to keep track of objects being replicated and existing references among them. There are other data structures private to each module that will be presented when each module is described in detail, in the following parts of this document.

Summary of Chapter: In this chapter we presented the architecture of OBIWAN. We introduced a high-level description of its application model and network architecture. We briefly described each of the OBIWAN middleware components and the fundamental data-structures that support its main aspects: i) incremental object replication, ii) distributed garbage collection, and iii) adaptability. This architecture is inherent to the rest of the dissertation, where each of these aspects is addressed in detail.

II

Incremental Object Replication

(this page was intentionally left blank)

Replicants are like any other machine... – in "Blade Runner", Ridley Scott, adapted from "Do Androids Dream of Electric Sheep?", Philip K. Dick

Part II is dedicated to incremental object replication. Initially, we present the relevant related work concerning support for data sharing, broadly considered. Afterwards, we present a detailed architecture of the mechanisms provided by OBIWAN that allow portable, efficient, adaptable, and transparent (yet flexible) object replication.

Subsequently, a number of prototype implementations are presented (OBIWAN.Java, OBIWAN.Net, and M-OBIWAN). Special focus is also given to OBIWAN integration with application servers (OBI-Web), and support for object persistence (OBI-Per). Regarding compliance with commercial integrated development environments for object-oriented programming, we cover a Visual Studio plug-in (OBI-VS).

Later on, both qualitative (adequacy) and quantitative (performance) evaluations are presented (when meaningful) for each of the developed prototypes. This part of the dissertation ends with some conclusions regarding object replication.

(this page was intentionally left blank)

II.1 Related Work on Data Sharing

Information sharing through data sharing is a fundamental aspect to computer supported cooperative work (CSCW) (Greif 1988), and has been one of the main goals of distributed systems research. This has become even more so, recently, in the related fields of mobile, pervasive and ubiquitous computing. More and more people perform work and exchange data using their laptops, PDAs or mobile phones, even without being connected to a central network (e.g., using Wi-Fi (Crow et al. 1997) or Bluetooth (Haartsen et al. 2000)).

Support for data-sharing is a cross-cutting issue in system design. It must be taken into account at a variety of levels, with different design approaches concerning each one. Since this is a very broad field, we bound the analysis of existing work to those subjects and systems more closely related to our own.

This chapter is organized as follows. The next section briefly describes a taxonomy framework to characterize data-sharing systems, according to a number of main system design aspects (*vectors*). The following sections describe each vector, and as each one may be implemented in different manners (*alternatives*). For each alternative, we introduce some relevant related systems that opted for it in their design. Following its description, this taxonomy is also depicted in a summary table. Therefore, each system may be briefly described by a *signature* consisting of the vectors addressed (or not) and chosen alternatives. We then present a set of relevant related research systems and commercial technologies in greater detail.

II.1.1 Data Sharing Systems

To develop a systematization characterizing data-sharing systems, we analyze the following system design aspects or design vectors:

- System architecture.
- Programming model.
- Data-sharing model.
- Propagation of modifications.
- Portability and implementation-level issues.

We selected these design vectors because they are the ones more relevant to our work w.r.t. incremental object replication. There are several other important system design aspects, e.g., consistency, security, fault-tolerance, etc., that are outside the scope of this dissertation.

In particular, consistency enforcement is a very important subject in data-sharing, but orthogonal to this work. It has been extensively addressed in the literature (Barghouti and Kaiser 1991; Barbara 1999; Jing et al. 1999; Mascolo et al. 2002a; Serrano-Alvarado et al. 2004; Androutsellis-Theotokis and Spinellis 2004; Saito and Shapiro 2005). Consistency enforcement oversees/manages how data may be independently accessed and updated by participating nodes, while providing satisfactory guarantees about data recency (or *freshness*), w.r.t. applications and users. It is also responsible for avoiding or minimizing divergence of data and update conflicts.

System architecture deals with the organization, deployment, and role of participating nodes of the system, as well as the nature of their interactions.

Programming model focuses on how shared-data is represented, and structured. It also describes the kinds of operations that can be performed by the applications accessing and manipulating it.

Data-sharing model describes how the shared-data is effectively made available to applications running on participating nodes. It determines if there can be several instances of shared-data items, and if so, the qualitative nature of those different instances. It also deals with the policies and mechanisms managing the creation, deployment and destruction of the different instances.

Propagation of modifications defines how modifications performed by applications on shared-data are represented and made available to other participating nodes, and how this is influenced by extracting semantic information associated with data, application and/or users.

Portability describes to what extent users and application developers must accommodate changes to their chosen operating and development environment (namely w.r.t. the prescribed programming model), in order to use and develop applications targeting the system.

Table II.1.1 presents the system design vectors and all their corresponding alternatives. Choosing certain alternatives in one vector may also frequently lead to specific alternatives in other vectors, e.g., mobile agents and migration.

II.1.1.1 System Architecture

Data sharing may be supported in different ways. Participating nodes may be organized and interact in different manners. We distinguish the following alternatives w.r.t. system architecture:

- Centralized server or data repository.
- Generalized client-server model.
- Peer-to-peer model.

Vector	**Alternative**
System Architecture	Centralized Client-Server Peer-to-Peer
Programming Model	Files Tuples Databases Objects Components Mobile Agents Structured HTML/XML files
Data-Sharing Model	Remote Execution Publish-Subscribe Migration Caching Replication
Propagation of Modifications	Update-based Operation-based Hybrid
Portability/Implementation Issues	Operating system. Virtual machine. Application binary code. Application source code. API replacement. Special-purpose programming language.

Table II.1.1: System design vectors and alternatives.

Centralized: The most traditional and less flexible architecture is based on having a logically *centralized* server or data repository. There may be a number of other participating nodes, but they all behave as clients of this central server. There is a tight association to the server, as clients usually connect to the same server, where all the shared-data is ultimately stored, or all the functionality (services offered) resides. Although logically centralized, the server may be physically replicated for performance and fault-tolerance reasons. It needs not be a single machine. Common examples of systems following this architecture include commercial database servers (Date 1999), holding data that is accessed and modified by several client nodes running applications.

Client-Server: A generalized *client-server* architecture (Sinha 1992) is more flexible than a centralized server approach. It still maintains a clear difference between client nodes that mainly address human interaction, and server nodes that store shared-data and/or provide services. Nevertheless, with this architecture, the complete data-space and functionality do not reside in a single server. Instead, they may be spread over a number of servers. Partition of data and services may obey to several criteria (e.g., geographical distribution, logical data structure, and organizational/institutional ownership of data). Naturally, there may be alternative servers for the same data and services due to the performance and availability reasons. In a typical computation, a client may contact any number of these servers to obtain data, execute some task with it, and store results. In a client-server architecture, to share data among clients, it must always be performed by intermediation of one or more servers. Clients may perform exclusively user-interaction (thin-client), or include some application functionality (full/fat-client). Web applications, developed and deployed following a three-tiered approach, are a common example of a generalized client-server model with multiple clients and multiple servers.

Peer-to-Peer: Since the designation *peer-to-peer* is used rather broadly in the literature, we define peer-to-peer (Androutsellis-Theotokis and Spinellis 2004) architecture primarily as one in which any participating node may behave, both as a client and/or as a server (even simultaneously). Thus, a peer-to-peer approach is generally more flexible than a client-server one, since each participating node is able to interact with any of the other nodes. Peers, in principle, have equal ability to store data and offer services to others. Every one cooperates by storing shared-data, providing services, or locating other peers and routing communication. However, in practice, different resource capabilities of nodes determine that this contribution will be unbalanced (e.g., super or ultra-peer nodes that store and serve higher volumes of data than *regular peers*). Systems with peer-to-peer architectures may also incorporate the ability to actively self-organize their network topology as participating nodes enter and leave the system, or to variations in connectivity.

Additionally, systems may have only a fraction of nodes carrying both client and server code, while others more restricted in resources (e.g., PDAs, mobile-phones) behave mainly (or solely) as clients. This way, some nodes appear as peers to other peer nodes and as typical servers to limited client-only nodes. These systems can also be considered following a peer-to-peer architecture, or a hybrid of client-server and peer-to-peer architectures. A typical example is the nomadic scenario for mobile and pervasive computing.

Traditionally, computational grid infrastructures (Foster and Kesselman 1997; Foster and Kesselman 1998; Baker et al. 2002) have been deployed according to a client-server architecture. Presently, however, they have been incorporating more and more traits of peer-to-peer architectures, assuming also a hybrid nature. In these systems, client nodes, operated by users, submit jobs to specific server machines (grid controllers, schedulers, etc.) that, in turn, negotiate with other peers (normally representing their own private clusters) where to deploy jobs, tasks, resources and store results (Talia and Trunfio 2003; Pallickara and Fox 2003; Andrade et al. 2005).

II.1.1.2 Programming Model

Applications can be developed according to several different programming paradigms that define the structure of shared-items and the operations that may be performed on them. Many of these paradigms were originally defined without distribution in mind. Since then, several have been extended to mobile and distributed scenarios. Such extensions are provided by the middleware. Thus, there are different abstractions supported by the underlying mobile middleware that, accordingly, provide the corresponding application programming interface (API). For example, the middleware may simply provide a file system API, or it may support more complex abstractions such as tuples, relational entities, objects, or components.

One relevant characteristic of such paradigms is their ability, or lack of it, to support arbitrary graphs of data. As a matter of fact, the existence of data graphs has a strong influence in how data is accessed and manipulated and, thus, how it must be shared. This programming model in which applications handle arbitrary data graphs is most widely used and is highly flexible.

The object-oriented paradigm naturally supports such notion of data graphs; the same applies to files whose contents include references to other files (e.g., graphs of HTML files connected by URLs), i.e., structured files. As already stated in Part I, the work presented in this dissertation focuses on object-oriented programming, mainly, and structured HTML/XML files.

Nonetheless, for completeness, in the rest of chapter we also address the data representations and semantics adopted by several relevant projects in the area of mobile and distributed computing. For instance, we also consider the file model because file system support is widespread and is well known both by users and application programmers. In this model the mobile middleware offers a file-based API (extended with specific functionalities for file-sharing) in which there are no references between files.

Data used by applications can be structured in various ways. These alternatives in structure reflect, and simultaneously determine, the different sets of operations, as well as their properties, that can be performed on data.

Generally, the more structured the data is, the more sophisticated are the operations that can be performed on it. Furthermore, these operations may be described with higher levels of abstraction and expressiveness.

We now present the most relevant data representations used and their semantics. They are introduced by increasing order of expressiveness, from flat files to XML documents. Likewise, possible operations range from hard-coded file read, write and seek operations, to rich transformations performed in XML documents, possibly specified in declarative form. We refer to several relevant systems that have been developed in the past supporting the data-sharing with the following data representations and semantics:

- Files.
- Tuples.
- Databases.
- Objects.
- Components.
- Mobile Agents.
- Structured HTML/XML files.

II.1.1.2.1 Files

Files, as far as the operating system is concerned, are flat in structure. Their format is managed by functionality hard-coded into applications or libraries. Therefore, applications are limited to using file system primitives such as read and write (either buffered or random) and seek.

There is no built-in support for inner data structures. The main advantage of using files for data-sharing stems also from its simplicity and availability in almost any operating system.

Earlier efforts in distributed file-systems include Network File System (NFS) (Network Working Group 1989; S. Shepler 2003), and Andrew File System (AFS) (Howard et al. 1988). Present commercial products include the Common Internet File System (CIFS) (Leach and Perry 1996; Hertel and Hertel 2003). CVS (Cederqvist et al. 2002), although based on a central repository, is a widely used system that allows version control, for collaborative file edition, in distributed environments.

Relevant projects of mobile and distributed computing support for file sharing include CODA (Kistler and Satyanarayanan 1992; Satyanarayanan 2002), which was the first to address the issue of disconnected work in distributed file systems. Further work has been devoted in Odyssey-CMU (Kumar and Satyanarayanan 1993; Noble et al. 1995; Noble et al. 1997), concerning application adaptability in mobile environments.

Ficus (Popek et al. 1990), Rumor (Guy et al. 1999), and Roam-UCLA (Ratner et al. 2001; Ratner et al. 2004) also represent a line of very interesting work concerning distributed file systems with growing concerns w.r.t. mobility. Other recent work on file systems for mobile environments includes Rufis (Shapiro et al. 2004) and FEW (Preguiça et al. 2005).

OceanStore (Kubiatowicz et al. 2000; Rhea et al. 2003) is an example of a large-scale distributed storage system, employing a peer-to-peer architecture. The objects stored are regarded as opaque files, divided in data-blocks (Rhea et al. 2003).

II.1.1.2.2 Tuples

Tuples are aggregation records of a variable number of fields which, originally, could only store strings and numerical values. Tuples offer a structured approach based on a sound model for distributed programming. Tuples are stored in a shared tuple-space that may span several nodes, merging concepts from shared memory and message passing systems. The operations performed on tuples are insertion(*out*), read(*rd*), destructive read or tuple consumption(*in*), and live tuple creation(*eval*) that dynamically creates a tuple from results returned from spawned processes.

The main advantage of using tuples for distributed application programming is the inherent structure provided while maintaining simplicity. In addition, use of insertion and consumption operations frequently obviates the need for locks in variables. Nevertheless, tuple spaces are not ordered. If several tuples match the template provided in *rd/in* operation, there are no guarantees which one will be fetched. Hypothetical references among tuples are not explicit, only interpreted as such by applications. Therefore, referential integrity is not upheld.

Tuples were originally introduced in Linda (Gelernter 1985; Ahuja et al. 1986; Carriero and Gelernter 1986). More recently, its principles have been adopted in Jini JavaSpaces (Eric Freeman 1999), and in TSpaces (Wyckoff 1998), supporting indexable fields for optimized searching. Tuples are also used in Java (van Reeuwijk and Sips 2002; van Reeuwijk and Sips 2005), extending the language syntax, to avoid the memory overhead associated with class instances in Java(e.g., references to class and synchronization monitor in all objects), when implementing even simple data-structures.

Tuple systems for mobile and distributed computing include Limbo (Davies et al. 1997) and L2imbo (Davies et al. 1998a; Davies et al. 1998b), Tuples On The Air (Mamei et al. 2003), One.World (Grimm et al. 2000; Lemar 2001; Grimm et al. 2004), Limone (Fok et al. 2004) and are announced in Enterprise TSpaces (IBM 2003). Work described in (Liskov 1989; Patterson et al. 1993) is focused on fault-tolerance for tuple spaces.

II.1.1.2.3 Relational Databases

Broadly defined, the term database can be be applied to any data sharing system based exclusively on query and update operations, regardless of data structure. A database may then be just a flat collection of opaque items, attainable via explicit identification (e.g., item name, item unique-ID).

Systems that do not target a specific programming model, and that present an architecture independent of data structure and APIs used (i.e., application-agnostic), are sometimes presented either as database systems, or file and object-based. Nonetheless, we include systems in the database programming-model when neither a file-system API, nor specific support for object-orientation and graph navigation, is implied (e.g., IceCube (Kermarrec et al. 2001; Preguiça et al. 2003b), and Deno (Keleher and Çetintemel 2000; Çetintemel et al. 2003)).

Since operations on database data may be tagged as query/read and update/write, consistency and reconciliation mechanisms can be defined independently of data representation. An interesting family of databases is that of relational databases. Relational databases comprise structured tables containing records with pre-defined number and type of fields. They provide a more structured data space than tuples, following relational algebra.

For relational databases in particular, operations on data are declaratively defined by SQL-queries for insertion, update and removal of records. Data on different tables can be joined by matching field values. Referential integrity, uniqueness and record ordering is provided. Several queries can be composed into transactions guaranteeing ACID properties (Gray and Reuter 1993) that may be relaxed in order to provide extra flexibility, e.g., in mobile environments (Lu and Satyanarayanan 1995).

While relational entities can refer to each other by means of foreign keys, thus creating a graph, applications usually access such data by performing queries. Thus, there is no navigation on such a graph as it happens with object-based applications.

One influential work regarding data-sharing is Bayou (Demers et al. 1994; Terry et al. 1995; Petersen et al. 1997; Terry et al. 1998), that addresses the merging of concurrent updates performed during disconnection periods. Relational databases in mobile computing have also been addressed in Mobisnap (Preguiça et al. 2003), and SQLIceCube (Preguiça et al. 2003a).

II.1.1.2.4 Objects

Object-orientation is the most widely used approach aimed at providing a structured and type-safe data representation, within programming languages. Objects contain typed fields and properties. Class and interface types define the methods that can be invoked on an object. Objects can contain references to other objects. This allows the creation of object graphs. Applications essentially navigate in the object graphs, starting from some root object and transversing references to other objects. Object-orientation allows type inheritance (with and without implementation inheritance), polymorphism, aggregation and composition (Brooch 1993).

Operations performed on objects are frequently expressed in high-level languages such as Java (Arnold and Gosling 1996) and C# (Archer 2002), for example. Besides regular constructs from imperative languages (e.g., selection blocks, loops), object-oriented languages provide object instantiation, reference manipulation and method invocations. Furthermore, some object-oriented languages provide reflection mechanisms that allow inspection of type and value information. This information is associated to fields, properties, methods, classes, etc., and to various extents, reflection allows their manipulation and modification.

This is arguably the most widely adopted programming model. Naturally, object-orientation in mobile and distributed computing has been addressed in several research projects such as Thor (Liskov et al. 1992; Gruber et al. 1994; Liskov et al. 1999), Larchant (Ferreira and Shapiro 1995), PerDiS (Ferreira et al. 2000), OBIWAN (Veiga and Ferreira 2002a; Ferreira et al. 2003), M-OBIWAN (Veiga et al. 2004; Santos et al. 2004), DERMI (Pairot et al. 2004), Javanaise (Caughey et al. 2000; Hagimont and Boyer 2001), Gold-Rush (Butrico et al. 1997), Alice (Haahr et al. 2000), Pro-Active (Baduel et al. 2002), Aroma (Narasimhan et al. ; Narasimhan et al. 2001), the ORCA shared-object system (Bal and Tanenbaum 1990; Bal et al. 1992), and Manta (Maassen et al. 2000), among others. However, not all have addressed with the same level of concern the challenges raised by mobility environments.

Today, the most relevant technologies addressing object-oriented programming in distributed systems are Java RMI (Wollrath et al. 1996), .Net Remoting (McLean et al. 2002), and SOAP (Box et al. 2000).

Object-orientation may also be combined with concepts originating from the database world. This is the case with object-oriented databases (Atkinson et al. 1989; Zdonik and Maier 1990), and shared-objects with transactions. Object-oriented Databases are simultaneously database systems (providing a query language, persistence) and object-based systems (enabling navigation through object graphs, type inheritance, polymorphism, etc.). Earlier examples include Exodus (Carey and DeWitt 1986), O2 (Lecluse et al. 1988; Deux et al. 1990; Deux et al. 1991), Gemstone (Butterwoth et al. 1991). Examples of recent work include Ozone (Braeutigam et al. 2002) and DB4O (db4objects, Inc. ; Paterson et al. 2006).

Persistence in object-oriented systems may also be achieved, with less flexibility, by leveraging existing relational databases, employing object-relational mapping. Examples include OJB (Apache Foundation 2002), Hibernate (Iverson 2004), and implementations of the JDO (Java Data Objects) (Russel 2002; Russel 2003) and JDO 2 (Russel 2005; Russel 2006) specifications. Microsoft .Net ObjectSpaces (Esposito 2004) and LINQ.Net (Box and Hejlsberg 2006) are related proposals for object persistence in Windows platforms.

II.1.1.2.5 Components

Programming based on components can be regarded as the next step in object-orientation. A component is an aggregation, wrapping several objects that are functionally related for a certain task. In this model there are no freely referenced objects across components. Graph navigation is restricted to within the component.

With components, the development of applications and services can be assisted by a set of tools that enable component combination. The runtime should be able to associate (bind) compatible components so that they work together. For this, components must be registered and expose some form of manifest, stating what they perform, possible dependencies, and how they can be combined.

An influential work in component-based data-sharing is that of the Rover Toolkit (Joseph et al. 1995; Joseph et al. 1997). In Rover, relocatable data objects (RDO), can be regarded as components, since they typically expose specific functionality, possibly encapsulating a private graph of objects. Earlier work includes SOS (Makpangou and Shapiro 1988), where applications are structured around *fragmented-objects*, components defined by a specific language (FOG).

Components have also been addressed in Roam-DoCoMo Labs (hua Chu et al. 2004), and SyD (Prasad et al. 2004). In DOORS (Preguiça et al. 2001), components are called *coobjects*, which are pre-defined aggregations of other data objects, and responsible for a specific part of the application functionality.

Components have been proposed as the basis of present *enterprise computing*, e.g., CORBA Component Model (Object Management Group 2002), Enterprise Java Beans (Malena and Hapner 1999; EJB 3.0 Expert Group 2006), OSGi Framework (Alliance 2003), and .Net with COM+ (Platt 1999; Löwy 2001; McKeown 2003).

II.1.1.2.6 Mobile Agents

The concept of *agent* is used in the literature in quite an encompassing fashion. Its meanings range from plain mobile code (classes or code snippets), to sophisticated software entities equipped with artificial intelligence capabilities (www.fipa.org 2002) that may act on behalf of human users in online interactions/transactions.

In the context of this dissertation a mobile agent is a software component (carrying code and data, the agent's state) that is extended with a private control thread. Optionally, an agent instead of carrying code may contain only a reference to a location where its code can be downloaded from. Mobile agents must run inside a agent execution environment deployed in each host system. Agent execution can be suspended and later resumed. Mobile agents may move across nodes, either decided by the system or autonomously. This implies suspending agent execution, transferring code, data, and execution context to another node, and resuming agent execution on arrival.

There is a great number of proposed mobile-agent systems (Rodrigues da Silva et al. 2001) such as Voyager (ObjectSpace 1997), Aglets (Lange and Oshima 1998), General Magic's

Odyssey (General Magic 1997), Lime (Murphy et al. 2001), Mars (Cabri et al. 2000), and Agent-Tcl (Gray 1995).

II.1.1.2.7 Structured HTML/XML Documents

Structured documents such as XML (W3C-XML ; Eckstein 2001) and HTML (W3C-HTML ; Niederst 1999) files are the data representation with the most implementation and presentation independence. These representations are heavily based on influential work developed in the context of SGML (ISO 1986).

They have an internal structure based on hierarchical nodes, possibly with references to nodes or other documents. This structure is exposed to applications reading (parsing) these files. One of the main advantages of these data representations is that they are text-file based, easing portability to any platform.

The usage of HTML documents is tightly related to web browsing and content producing. XML, however, can be used for data representation in any environment and for any application.

Applications can navigate on graphs of HTML documents (a common task in web browsing) through URLs referencing other documents. With XML documents, operations can be expressed in XPATH, XPointer, and XSLT, allowing attribute query and update, navigation, and content transformation (Tennison 2001).

Sharing of web content, besides web caching (see Section II.1.1.3.4) and plain mirroring of web sites, has been addressed in LOCKS (Rosenthal and Reich 2000; Reich and Rosenthal 2001; Maniatis et al. 2003; Maniatis et al. 2005), RepWeb (Veiga and Ferreira 2003c), (Veiga and Ferreira 2004b) (taking into account dynamically generated content), and in Replets (Zhou et al. 2004) (regarding servlet objects). XML data-sharing has been addressed in XMIDDLE (Capra et al. 2001; Mascolo et al. 2001; Mascolo et al. 2002b).

II.1.1.2.8 Summary

Every data representation has one or a number of fields of application where it is most suited. However, due to their relatively broader adoption, there are two programming models that with greater impact: i) the object model, due to its wide acceptance in almost every aspect of application programming and, ii) the file model, since it provides the most basic abstraction and most systems support it. However, in this dissertation, we only address the object model and structured documents.

II.1.1.3 Data-Sharing Model

Applications can access shared data and services, provided by different nodes, according to several different models. Key issues regarding a data-sharing model are co-location of data and code, and placement of data in nodes that perform interaction with application users. We discuss the following data-sharing models that may be provided to applications and users:

- Remote invocation.
- Publish-subscribe.
- Migration.
- Caching.
- Replication.

Note that, however, nothing prevents a specific realization of a particular data-sharing model from using a different model in its underlying implementation, e.g., using remote execution to support data replication.

The choice of a particular data-sharing model has implications in several aspects such as performance, communication, availability, data recency/freshness, resource demand and consumption.

II.1.1.3.1 Remote Invocation

In this data-sharing model, applications running on a node cannot have direct access to data located at other participating nodes. Remote execution is the most traditional data-sharing model. It has long been adopted by several systems with centralized and client-server architectures. It is based on the notions of stub (Birrell and Nelson 1984) and proxy (Shapiro 1986) for transparency.

Several programming models have been extended to support remote execution. In models based on imperative languages, remote execution is referred as Remote Procedure Call (RPC), such as Sun/ONC-RPC (R. Srinivasan 1995), while in object-oriented programming, it is commonly referred as Remote Method Invocation. Initial research work on this area includes SOS (Makpangou and Shapiro 1988), Soul (Shapiro 1991b), and Network Objects (Birrell et al. 1993b). Java RMI (Wollrath et al. 1996)[1], .Net Remoting (McLean et al. 2002), already mentioned, are leading technologies w.r.t. commercial products. Earlier efforts include DCE (Leser 1992), CORBA (Siegel 1996), and DCOM (Sessions 1998).

More recently, XML has also been used to provide inter-operability among participating nodes with different execution environments (operating system, virtual machine, programming language), in the context of Web Services, e.g., XML-RPC (Winer 1999; Udell 1999) and SOAP (Box et al. 2000).

In database systems, remote execution is performed by Stored Procedures, that may be coded in SQL, or in an object-oriented language (Java (Arnold and Gosling 1996), C# (Archer 2002)). In Ozone (Braeutigam et al. 2002), client applications always invoke persistent objects by means of remote invocation (Java RMI).

Remote execution has been adapted to address frequent disconnection scenarios. This is achieved using queued RPC invocations, as proposed in Rover (Joseph et al. 1997).

[1]Remote Method Invocation.

DERMI (Pairot et al. 2004) provides remote method invocation for objects deployed over P2P overlays.

II.1.1.3.2 Publish-subscribe

In the Publish-subscribe (Eugster et al. 2003) data-sharing model, there are three kinds of data: advertisements, subscriptions, and publications of events (pub-sub is also related to event-based programming). An event is simply a notification that something interesting to the system, applications or users, has occurred. In this model, data items are immutable, i.e., once created they cannot be modified, only consumed and discarded.

Events may be identified by name, type/category, or additional meta-information. Events may also contain time information and additional application-specific data. Participating nodes may behave as publishers or subscribers for specific events.

Initially, publishers may send other participating nodes *advertisements* of the events they publish. Of the latter, those that have interest in being notified of future occurrences of the event, may *subscribe* to it, i.e., register their interest. When the publisher determines that a specific event has occurred, it sends (*publishes*) notifications of the event to the subscribing nodes.

With respect to programming model, event message-data and processing may be based on arbitrary text or binary content, object-based data, XML-based data, or invocations (such as callbacks).

Event subscription can be performed based on a variety of aspects, namely topic/subject (plain text or using XPATH expressions), object type, content (presence of specific properties or conformity to an object template), predicate function. Nodes may subscribe to composite events (Bacon et al. 1995; Bacon et al. 2000; Li and Jacobsen 2005), i.e., subscribe to combinations of distinct events to be notified only when the composite events occur.

Data dissemination and storage in pub-sub systems may be realized resorting to any of the proposed system architectures: i) centralized in user interface design, ii) client-server, or iii) peer-to-peer. Subscription of events must be recorded; it may be on the subscriber, the publisher (e.g., callbacks), on broking servers, or spread across the network as in peer-to-peer architectures.

An evaluation criteria of publish-subscribe systems is to what extent they allow publishers and subscribers to be logically decoupled in terms of time, space, and/or synchronization.

Examples of distributed publish-subscribe systems include work in (Bacon et al. 1995; Hayton et al. 1996) and, more recently, Jedi (Cugola et al. 2001), Siena (Carzaniga et al. 2000), ToPSS (Petrovic et al. 2003) and PADRES (Fidler et al. 2005). Publish-subscribe in mobile environments has been addressed in (Huang and Garcia-Molina 2004).

Scribe (Castro et al. 2002) performs event notification on P2P overlays, and is further extended for publish-subscribe in (Tam et al. 2003). DERMI (Pairot et al. 2004) is an example of an object-based middleware on top of a peer-to-peer pub-sub system, upon which remote invocations are modeled. Tuple-based systems with asynchronous notifications also follow a pub-sub data-sharing model.

II.1.1.3.3 Migration

Migration is a data-sharing model that allows data items to be transferred across different nodes, so that they become closer to better available resources, other related data items, or for interaction with users. With migration, despite data transfer being essentially a copying operation, there is no change in the number of instances of the shared-data item. Thus, at transfer, it is copied to a destination node and eliminated in the source node.

Migration is extensively used in conjunction with the mobile agent programming model, for agent wandering, or simply for task migration.

Migration is also used with component programming model. It is useful to improve availability and scalability. In Rover components may migrate from server nodes to client nodes prior to impending disconnection, to improve availability. Components may be migrated among application servers to perform load-balancing (Frenot et al. 2002; Kapitza et al. 2005) for better scalability, namely in enterprise computing.

The two following data-sharing models allow the existence of several instances of the same shared-data item. However, there are some differences that will be made clear next.

II.1.1.3.4 Caching

Caching is a data-sharing model that allows and effectively leverages the existence of several instances of the same data item, placed at different nodes. One instance is referred to as original, while the others as cached copies. These instances continue to embody a single logical data-item that is copied solely to facilitate sharing. This avoids the performance, latency and availability penalties that applications might incur from using, for instance, a remote-execution model to access a shared data item. Caching has been traditionally employed in systems with centralized server or client-server architectures.

A cached copy of a shared-data item does not enjoy the same qualitative nature of the original. The original resides at a specific node, called home node, and is preserved until explicitly deleted. Cached copies at other nodes, however, may be temporary with a specific time duration, may be silently discarded (forwarding to the original), subject of replacement policies, etc.

Caching has been extensively used in web systems (Wang 1999; Rodriguez et al. 2001; Iyer et al. 2002), to accelerate download. It is performed by groups of cooperating servers, possibly organized hierarchically.

An important distributed system with object caching is Thor. This system provides a hybrid and adaptive caching mechanism handling both pages and objects. In Thor, a number of server nodes hold a distributed object graph. Thus, objects reside at servers and are cached at clients only for increased performance and reduced latency. Cached copies at clients cannot be further copied elsewhere, except to be returned to its home server when updated.

Most object-oriented databases (OODBs) (Zdonik and Maier 1990), for example such as O2 (Deux et al. 1990; Deux et al. 1991), GemStone (Butterwoth et al. 1991), employ some form of caching. There has been some work on object caching in CORBA (Kordale et al. 1996) as well. Cascade (Chockler et al. 2000) proposes an hierarchical cache system for CORBA objects.

II.1.1.3.5 Replication

Data replication also makes use of several instances of the same data-item. However, as opposed to caching, with replication, each instance of a data-item (a *replica*) is not transitory and is in fact considered, as a first-class entity in the system. A replica should only be discarded when all replicas of the shared-item are considered useless. Nonetheless, in some cases, one of the replicas may be considered as a master replica, namely for consistency purposes. Replicas may also create other replicas of their data to be handed to other nodes.

Locally replicated data is always readily available to applications (even when the network is down), with access time orders of magnitude lower than non-local data, and avoids frequent, and possibly lengthy and costly connections to the underlying network (specially so in the case mobile devices with GSM, CDMA or GPRS connectivity).

Chapter I.1 has already presented the main arguments for data replication. It has been a prime model adopted for data-sharing. It improves availability, performance, and cost-effectiveness. Data replication has been applied to client-server architectures and is fundamental in peer-to-peer systems.

Data replication has been used with virtually almost every programming model. It has been comprehensively addressed in various projects and systems, such as: CODA, Ficus, Rumor, Roam-UCLA, Rufis, Few, One.World, Enterprise-Tspaces, Bayou, Deno, IceCube, OceanStore, Mobisnap, SQLIceCube, Javanaise, GoldRush, Larchant, Perdis, OBIWAN, DERMI, replicated CORBA, LOCKSS, XMIDDLE, Replets, RepWeb. Systems based on Distributed Shared Memory (Li and Hudak 1986; Li and Hudak 1989) systems also make use of replication.

Replication of RMI Server Objects has been studied in (Baratloo et al. 1998), while replication of components in the context of Java-based enterprise computing has been covered in (Kistijantoro et al. 2003).

II.1.1.3.6 Hybrid approaches

Systems may use a hybrid of data-sharing models. They can use different data-sharing models for different types of shared-items (e.g., migration for mobile agents and replication for private data the agent may hold), or switch between data-sharing models (e.g., when connected, switch from caching, migration or replication to remote-execution model). They may also use a number of data-sharing models in combination (e.g., migration with caching or replication). DERMI provides migration, caching and replication.

In Oceanstore, replicas of shared-data items are maintained in several nodes for increased availability and proximity to client nodes. Additionally, shared data is cached in several, or all intermediary nodes in the network, as it passes through them. Oceanstore performs introspective replica management. Replicas may be migrated to different nodes to accommodate demand.

II.1.1.4 Propagation of Modifications

There are the two main families of approaches to represent and propagate modifications performed on shared-data. These are particularly relevant in the context of caching and replication data-sharing models. Propagation of modifications can be:

- State-based.
- Operation-based.

The state-based approach propagates changes made to shared-data simply by providing the new content of the shared data after modification.

The operation-based approach propagates modifications as a sequence that describes the relevant operations that were performed on the data. Remote invocation is a data-sharing model that adopts an operation-based approach to propagate modifications to (remote) data.

The actual representation of these operations may be dependent of the programming model used. The greater this dependency is, the greater is the semantic information provided with the operations. Nonetheless, an application-agnostic approach simply regards *read* and *write* operations.

State-based approaches have the advantage of being easier to implement, mostly non-intrusive to applications, and several modifications can be combined in a single update. However, state-based approaches are vulnerable to small modifications performed on large objects. These difficulties can be mitigated by sending only the differences between new and previous values, possibly compressed.

Operation-based approaches are more expressive, in the sense that some broad or scattered modifications on data can be represented very efficiently as operations (e.g., inserting scattered sentences in a very large text document). Nevertheless, they require application instrumentation in order to record (i.e., logging) the operations performed, or that these applications are already operation-based (e.g., SQL databases, file system directory maintenance).

State-based approaches are more suited to systems based on sophisticated programming languages where *representative* operations (i.e., semantics associated with each modification) may be very difficult or impossible to extract correctly. Furthermore, they are also preferable in systems where it may be prohibitive to log every operation performed (e.g., an object-oriented language program may perform hundreds of interdependent read and write operations on a graph of objects for common tasks that include graph transversion, search, and modification).

Operation-based approaches are widely used in systems where users perform common interactions with a well-defined semantics (file-systems, meeting scheduling, document edition, stock management, etc.). This approach also provides interesting properties in the context of consistency (e.g., operation changing, re-ordering).

The two approaches can be combined into hybrids. A node may store a short history of differences between original and modified data (i.e., *diffs*), that may be transmitted instead of the new state, where a common previous (but recent) state is known by both. Several *diffs* may be applied in sequence. In certain conditions, operations performed may be extracted from the analysis and comparison of two versions of a shared-data item (e.g., extracting insertion, modification and deletion of lines in text files).

In systems based on remote-execution, propagation of modifications is inherently operation-based. In Publish-subscribe systems, shared-data is mostly read-only. Publications are just consumed by subscribers, while modifications to subscriptions are propagated in operation-based manner.

In systems based on migration, caching and/or replication, modifications can be propagated in a directed manner (e.g., sending modifications to the nodes where shared data was originally obtained from, or to other alternate nodes), or indirectly as in epidemic (Petersen et al. 1997) propagation (where modifications are exchanged in pair-wise anti-entropy sessions when two nodes meet).

Relevant examples of systems that propagate modifications to shared-data as operations include Bayou, IceCube and Deno. On the other hand, Ficus, Roam, Thor, OBIWAN, and web caching systems are examples of systems that propagate modifications to shared-data by sending the content of updated state. Coda uses a state-based approach for modifications on files, and an operation-based approach for directory maintenance operations.

II.1.1.5 Portability and Implementation Issues

Every system, while addressing each (not necessarily every) vector in system design by employing a certain alternative, must interact ultimately with the computing environment in nodes. Often, this requires modifications to the environment. Such modifications may surface at a number of levels, with variable visibility. We highlight the following:

- Operating system.
- Virtual machine.
- Application binary code.
- Application source code.
- API replacement.
- Special-purpose programming language.

These modifications may manifest themselves to node administrators, application developers and application users. These categories may overlap, e.g., for a PDA, application user and node administrator are the same person.

Operating System: The lowest level that may require modification is the operating system. This is needed when a system requires that operating system functionality be extended. This may involve recompilation of part of the operating system. Alternatively, the OS may be extended by installation of new modules. For instance, file-system extension (e.g., to enable caching and replication) in Windows platforms may be deployed via Installable File Systems.

Distributed shared memory systems require an extended page-fault handling mechanism capable of fetching pages from the network. This mechanism is also used in pointer *swizzling* and orthogonal persistence for programming languages (Wilson 1990; White and Dewitt 1992; Wilson and Kakkad 1992; Hosking and Moss 1993; Sousa et al. 1993).

Modifying the operating system has several disadvantages: it requires administration privileges and the effort must be done for every operating system targeted by the system. Dependence on extension mechanisms provided by a specific operating system, although more portable and requiring less privileges, still prevents deployment on other operating systems.

Virtual Machine: An attractive alternative to changing the operating system is the use of a widely deployed virtual machine (VM), such as JVM (Lindholm and Yellin 1996),[2] or .Net CLR (Platt 2001).[3] Applications developed targeting a specific VM are obviously more portable, since they can be developed to a unified running environment, regardless of the different implementations of the VM w.r.t. different hardware, architectures and operating systems.

In fact, the virtual machine becomes the operating system, as far as applications and developers are concerned. Nonetheless, it may be necessary to extend the virtual machine itself, which raises issues similar to those of the previous alternative. The need to use a modified virtual machine is restrictive to application deployment. This is also the case when a system requires its own *non-standard* virtual machine, or running environment (e.g., Thor, GemStone, O2), since applications cannot be run elsewhere.

Application Code: Supporting a specific alternative of a design vector can also be achieved by extending application code. Code extension includes modification of application code as well as automatic generation of additional code (e.g., proxies) to be included in the application. This may or may not require access to the source code of applications.

W.r.t. binary code, some libraries used by applications may be replaced by customized versions that intercept function calls, adding some specific behavior. A typical approach is to replace part of the C-runtime library to intercept the invocation of specific functions (e.g., file and memory management).

Byte-code enhancement is a representative example of binary code extension. It is used in persistence frameworks for object-based systems, such as Hibernate, where it is performed at runtime when classes are first loaded, and in JDO reference and most compliant implementations, during compilation. Recent versions of OJB[4] also generate proxies using a byte-code manipulation library.

[2]Java Virtual Machine.
[3]Common Language Runtime.
[4]Version 1.0.4.

Code extension may also be performed on the source-code, i.e., via source-code augmentation. In OBIWAN, source-code is augmented to add object-fault handling and incremental replication. Binary code extension has the advantage of neither requiring access nor imposing modifications to source code. However, developers cannot see the modifications made to their code which may hinder the debugging process.

Automatic generation of additional application code may be performed directly (e.g., it has been used to generate RMI stubs and skeletons) or via source-code generation and ulterior compilation (e.g., proxies in OBIWAN, Ozone, and earlier versions of OJB, and stubs for C++, Java, and Smalltalk in Thor).

OJB and other object-relational mappers also extend application source code by generating SQL statements in order to read/write object contents from/to relational databases. In the specific context of applications written for Java, .Net, and CORBA, the use of reflective capabilities and techniques is invaluable in class analysis for code and behavior extension.

API and Programming Languages: The adoption of a given alternative for a design vector, in some systems, may also impose the use of a new application programming interface. This will allow additional flexibility and productivity in the development of new applications, at the expense of requiring porting existing applications that were not developed using the new API (e.g., Read-Write APIs from Bayou, Deno, and Pro-Active). This may be a time-consuming process, even with the aid of tools for automatic conversion.

In this alternative, we also include the tasks associated with supporting a specific API, such as developing dependency checks and merge procedures in Bayou, defining and extracting operations performed by applications in order to use IceCube, etc.

OJB and JDO implementations require the creation of XML files, containing mappings of class descriptions and associations that describe how and where object fields should be fetched from the database. In certain configurations, OJB requires classes to implement methods according to Javabeans calling convention (e.g., `getXXX, setXXX`) to populate object fields. As an alternative, it can use reflection to inspect and assign fields, with additional invocations and performance overhead.

In Hibernate, applications must be aware of the replacement of collections (e.g., List, Vector) of objects, with instances of a custom Collections API.

PERST (McObject 2003) requires that application classes derive explicitly from a common top-level class: `Persistent`. To prevent loading complete graphs into memory, PERST also requires the programmer's intervention by: i) overriding `recursiveLoading` methods, and ii) invoking `Persistent.load` explicitly.

DB4O, while entirely based on reflection, also requires explicit use of a specific API, in order to load additional objects from storage (using `db.activate(obj,depth)`). This is not required in *cascade* mode in which DB4O loads the entire reachability graph into memory, when an object is first accessed.

At the highest level, a system may impose the use of a specialized programming language, such as Theta for defining object structure and method code in Thor, and query languages in many object-oriented databases.

The level of intrusion is greater for system administrators on one end of the range (operating system) and greater for application developers on the other (special-purpose language). Desirably, this ensures that intrusion to users is kept to a limited degree. We believe that system dissemination will be favored if it is possible to minimize the two trends of intrusion. Arguably, this balance is optimized by using application code extension. Thus, middleware should not imply the modification of neither the operating systems nor the virtual machines, and should not impose radically new APIs.

II.1.1.6 Summary

In this section we presented a taxonomy framework for characterization of data-sharing systems, w.r.t. relevancy to our work. For each vector and related alternatives, we laid out its relevant features and offered examples of systems that adopt them.

Even though most combinations are possible, there are a number of dominant approaches in specific scenarios. More recent file-sharing systems are dominantly peer-to-peer, based on replication, propagate modified parts of files (*diffs*), and may replace the normal file-system API. High-end relational databases follow a client-server architecture, with centralized-storage (server), using SQL language for remote invocation of queries (while having some support for replication) that also represent modifications on data. The dominant approach in the world-wide-web is based on a client-server architecture, file-based, with several levels of caching (web-proxies), using the HTTP protocol (Network Working Group 1996). Modifications are propagated by updating complete files on proxies.

Mobile computing usually follows a hybrid of client-server and peer-to-peer architectures, and is based on either files, databases, objects, or components. Usually, replication or caching are employed to tolerate disconnection periods. Modifications can be propagated using an operation-based approach (e.g., for databases, and files (Preguiça et al. 2005)), or transferring updated-state (e.g., for objects and also file-based systems). Portability aspects range from extending operating system with drivers, to mandating the adoption of a specific API.

II.1.2 Case-Study of Relevant Systems

This section offers a case-study of some relevant projects related with this work. First, we describe some research projects from academia, including early influential works and more recent projects. Then, we characterize them against the proposed taxonomy in Table II.1.2 and analyze them w.r.t. the goals of this dissertation. Finally, we offer some insight on relevant commercial products, including pioneering efforts and current technologies.

II.1.2.1 Research Projects

II.1.2.1.1 Bayou

Bayou (Demers et al. 1994; Terry et al. 1995; Petersen et al. 1997; Terry et al. 1998) presents an architecture based on mobile-aware databases, used for data-sharing among mobile users, with high availability. It introduces the innovative notions of eventual consistency and epidemic propagation.

Each server contains a log of operations, and stable and tentative versions of the database state. Stable state reflects the execution of operations already committed. Tentative state reflects the execution of operations known by the server but not committed yet. It allows applications to make progress but those operations may still be rolled-back and re-done later, if necessary. Replicas are updated by epidemic propagation. When two servers meet, they exchange operations stored in their logs, appending the ones still unknown to each one.

Replica consistency is enforced by performing Write operations in the same, well-defined order at all servers. This achieves eventual consistency among servers, ensuring that all replicas will converge to equivalent states, although offering no guarantees on promptness. It employs a protocol that ensures stabilization even in the presence of conflicts. Application-specific conflict resolution is performed by special types of methods: dependency checks and merge procedures that are application-specific. A dependency check is run before an operation is applied, detecting possible conflicts. If it fails, a merge procedure is executed modifying the conflicting operation in a way that still serves the user's intended purpose (e.g., rescheduling a reunion at a later hour, or to a nearby room). Query and update operations, as well as dependency-checks and merge procedures can be coded in SQL.

Final decision on the ordering of operations, including conflict resolution, is performed by a designated server, the primary or home node. It may be a high-end server that is always available for a majority of users, or a user's laptop with priority over others. Every server must maintain a replica of the complete database. This may be unfeasible or undesirable with some mobile devices (Mascolo et al. 2002a).

II.1.2.1.2 Rover

The Rover (Joseph et al. 1995; Joseph et al. 1997) toolkit is a framework for the development of both mobile-transparent and mobile-aware applications. It addresses intermittent connectivity and bandwidth limitations in mobile environments, while optimizing communication among nodes. It introduces the notions of relocatable data objects (RDO) and queued remote procedure calls (Q-RPC).

RDOs are described as objects comprising data and code that can migrate both ways between clients and servers. Usually, RDO are more than plain objects since they may encapsulate large parts of application functionality. These are the main components to be defined by the developers of mobile applications (e.g., mail and news reader, web-proxy, mobile transparent file-system).

RDOs communicate among each other by means using Q-RPC. This allows applications to proceed, making non-blocking remote invocations, even in disconnection periods. All outgoing and incoming remote calls are logged. Request and reply of a single Q-RPC may be sent using different connections. Applications can inspect the progress of ongoing Q-RPC. Communication is optimized by a network scheduler that allows message batching, prioritization, and application-specific data compression. It minimizes energy consumption, delay, and communication cost.

II.1.2.1.3 Thor

Thor (Liskov et al. 1992; Gruber et al. 1994; Liskov et al. 1999) is a distributed object oriented database (OODB). Objects reside in one or more servers forming a distributed persistent heap. Applications act as clients of Thor. They cache objects locally and execute methods on them, within transactions. When transactions are committed, modified objects are sent back to their servers. Cached copies are created lazily. As object references are transversed, page-faults are triggered and handled.

When an object is cached, Thor caches the whole page containing the object. It employs a hybrid and adaptive caching (HAC) mechanism (Castro et al. 1997) that partitions the cache dynamically, handling both pages and objects. It retains pages with observed high locality throughout, otherwise it maintains only its most accessed objects.

Thor defines a format for objects in memory and their storage on file. Disk usage is optimized by means of a modified object buffer (MOB) that delays writing modified objects to disk. The MOB is flushed lazily as it fills up, writing several pages (i.e., segments) in a single disk operation. In addition, Thor provides a type-safe programming language (Theta), a concurrency scheme (CLOCC),[5] and a distributed garbage collector that manages client caches and the persistent store (referred in Part III.1).

II.1.2.1.4 IceCube

The IceCube (Kermarrec et al. 2001; Preguiça et al. 2003b) project aims at solving reconciliation of concurrent updates on different replicas, as an optimization problem. Applications running on clients manipulate replicated data and perform operations on it. Operation are logged and later sent to a generical reconciliation server, a centralized repository holding authoritative versions of all data items. IceCube performs reconciliation, by computing a schedule based on logged operations received from all replicas. Once completed, the calculated schedule is then sent to all other replicas to be replayed.

Scheduling is performed according to a dependence-graph that represents all the constrains among pairs of update operations, such as pre-conditions. Dependence-graphs combine constrains regarding data-types, applications, users, and system-wide. These constrains are application and data-type specific, and can be semantically rich, while the scheduler is independent

[5]Clock-based Lazy Optimistic Concurrency Control.

of any specific application. Examples of constrains include dependence, implication, and choice. In order to reduce the search space, IceCube also supports a commutativity relation among operations.

Scheduling is optimized using heuristics. When there are several possible correct schedules, the system ranks them, and selects the one with the highest utility (i.e., optimum), as defined by an application metric (e.g., maximize the number of committed operations, prioritize some types of actions).

II.1.2.1.5 Mobisnap

Mobisnap (Preguiça et al. 2003) is a database middleware system designed to transparently support applications running on mobile environments. It allows different clients concurrently updating the database by usage of mobile transactions, though modification of persistent data is only done at the central server (that acts as a primary replica).

Mobile transactions are expressed in unmodified PL/SQL, improving on previous results (Preguiça et al. 2000). It allows replication of relational-model data in the clients and semantically infers from client transactions, the necessary constraints (reservations) which, with a good degree of confidence, prevent conflicts and allow transaction completeness, independently, when these are replayed at the central server.

Reservations are a kind of semantic lock on data, associated with a time validity (lease). They are issued and enforced by a primary replica. They represent rights to use and/or modify data. Types of reservation include: *escrow* for data subject to division (e.g., stocks), *slot* for record insertion with pre-defined content, *value-change*, *value-use*, and *anti-lock* to prevent reservations w.r.t. the same records being issued to other clients.

Each replica maintains a tentative and committed version of the database, as in Bayou. Transactions completed at clients may be tentative-committed or reservation-committed. The former have completed successfully but there is no guarantee the same will happen when replayed at the primary replica. The latter ensure successful completion at the primary, provided they are replayed before the expiration of the lease.

II.1.2.1.6 Javanaise

Javanaise (Caughey et al. 2000; Hagimont and Boyer 2001) is a platform that aims at providing support for cooperative distributed applications on the internet. It caches objects in order to avoid costly remote method invocations, using application-dependent object clustering.

Application development in Javanaise is centered on the notion of *clusters* that are predefined aggregations of related objects. Classes of objects heading clusters must explicitly inherit, at least, from one of two interfaces: `CacheableCluster, PersistentCluster`. The programmer must be fully aware of clusters and is responsible for the decomposition of the distributed object graph into individual clusters. Nonetheless, clusters can be deployed dynamically.

Project	**System Architecture**	**Programming Model**	**Data-sharing Model**	**Propagation of Modifications**	**Portability**
Bayou	CS / P2P	database	replication	op-based (SQL)	Bayou API
Rover	CS	components	remote-invocation migration/caching	op-based (QRPC)	Rover API
Thor	CS	objects	caching	state-based (modified objects)	libraries (*veneer*)/ API and language (Theta)
Mobisnap	CS / centralized repository	database	replication	op-based (SQL)	extended db-engine
IceCube	CS / centralized repository	agnostic	replication	op-based	constrains API
Javanaise	CS	objects components	caching	state-based (modified objects)	cluster API

Table II.1.2: Design alternatives for related data-sharing research projects.

Local (intra-cluster) objects cannot be shared among different clusters. They are only known within their containing cluster and cannot be referenced from outside. Cluster objects mediate all inter-cluster references and invocations. Methods of cluster classes may only take as arguments, and return as results, references to other cluster objects. Clusters thus resemble components as RDOs in Rover.

Javanaise extends Java RMI with caching mechanisms. The application development follows a similar approach with some modifications in source code. A proxy generator is then used to generate indirection objects and a few system classes supporting a consistency protocol, e.g., single-writer-multiple-readers.

Analysis The design alternatives employed by the research systems described in the section are summarized in Table II.1.2. Taking into account the goals and challenges enunciated in Chapter I.1, we can also perform a comparative analysis of the systems presented, evaluating if and how projects address each one. We recall these goals and challenges in the form of the following requirements:

1. **Usage of Local Resources:** leverage the usage of existing local resources (CPU, memory) effectively.
2. **Support for Disconnected Work:** minimize dependency on the availability of network connection, e.g., by employing replication techniques.
3. **Transparent Support for Commercial OO Languages**: allow application development using widely adopted object-oriented languages (e.g., C++, Java, C#).
4. **Platform Portability:** impose modifications neither to operating systems nor to dominant commercially available virtual-machines (e.g., Java and .Net).
5. **Enforcement of Referential Integrity:** improve programming reliability by preventing dangling references.
6. **Adaptability**: provide mechanisms to control resource consumption on running nodes (e.g., memory, network).

None of the systems addresses all of the requirements. In some cases, one or more requirements were not even specifically targeted in their design. Nonetheless, they are obviously useful in their particular context. Table II.1.3 summarizes the results, indicating for each requirement, whether a system either: i) fulfills it completely (*Yes*), ii) addresses it partially (*Limited*), or iii) disregards it (*No*). To facilitate table readability, additional explanations are provided with individual footnotes below the table.

Project	**Usage of Local Resources**	**Support for Disconnected Work**	**Transparent Support for Commercial OO Languages**	**Platform Portability**	**Enforcement of Referential Integrity**	**Adaptability**
Bayou	Yes	Yes	No	Yes [a]	No	No
Rover	Yes	Yes	Limited [b]	Limited [c]	No [d]	Yes [e]
Thor	Yes	Limited [f]	Limited [g]	No [h]	Yes	Limited [i]
Mobisnap	Yes	Yes	No [j]	Yes [k]	Limited [l]	Yes [m]
IceCube	Yes	Yes	No [n]	Yes	No	No
Javanaise	Yes	Limited[o]	Limited [p]	Yes	No [q]	Yes [r]

[a]Posix and Java-based implementations.

[b]Imposes Rover API and RDO decomposition.

[c]Mobile-transparent File System for Rover Toolkit needs a OS-kernel module for client application.

[d]Does not enforce referential integrity among components.

[e]Provides pre-fetching and caching strategies, network optimization, and application-specific compression.

[f]Client requires that objects be already in cache and needs to be connected in order to receive invalidations.

[g]Both the structure of objects and their methods must be defined in Theta, a Thor-specific object-oriented language. Applications that make use of objects may be developed using stubs for C++, Smalltalk or Java.

[h]Thor provides its own virtual machine and its adaptation to JVM or .Net CLR would require modifying these virtual machines.

[i]Adapts the size of fetch-groups containing objects to be sent to the clients, based on cache hits of segments sent previously, without parameterizing neither for memory nor bandwidth usage.

[j]Applications are coded using extended PL-SQL.

[k]Java middleware on top of commercial RDBMS.

[l]Must be explicitly specified using SQL constrains.

[m]Clients replicate subsets of tables, i.e., possibly only a fraction of rows and columns or each table.

[n]Developer must specify relevant operations to the scheduler and log them.

[o]Requires connectivity to enforce pessimist consistency, e.g., single-writer-multiple-readers, in cached clusters.

[p]Requires developer to define static object clusters, and disallows references between objects internal to different clusters.

[q]Relies on Java RMI DGC that may drop referenced objects after lease expiration.

[r]Allows dynamic deployment of clusters.

Table II.1.3: Analysis of related data-sharing research projects w.r.t. the goals and challenges described in Chapter I.1.

II.1.2.2 Industry Standards and Commercial Products

In this section, we present a brief overview of object-oriented and component-based data-sharing platforms, commercially available. They are mainly focused on Java and .Net technologies.

II.1.2.2.1 Java

Support for data-sharing in early editions of the Java (Lindholm and Yellin 1996) platform was limited to object serialization, data-base connectivity (JDBC), and Java RMI (Wollrath et al.

1996). Serialization and JDBC allow applications to store and retrieve object graphs from persistent storage (files and relational databases, respectively) and over the network, while RMI allows distributed object invocation.

The Java Data Object specification (Russel 2002; Russel 2003) is a more evolved approach to data persistence. It is the corollary of earlier research and commercial projects (see Section II.1.2.2.3) in the fields of object-relational mapping, and object-oriented databases. It defines an API, containing a number of interfaces and classes, that application programmers should follow when defining classes whose instances are to be stored persistently. Furthermore, it describes how compliant implementations of the specification should rule the interaction between persistent storages and the Java runtime. This shields application programmers from dealing with the details of code related to persistence and the database.

In its initial version, JDO offered no recommendation on how to optimize bandwidth and memory usage, when client applications load objects from remote persistent stores. In its current version (Russel 2005; Russel 2006), JDO already includes support for partial loading of object graphs from the database, delayed (or lazy) object activation, and the definition of groups of objects that should be loaded collectively.

The Enterprise Edition of the Java platform (Shannon 2001; Shannon 2003; Shannon 2006) is a suite of specifications and API to support the execution of Java-based software in the context of large-scale enterprise systems. Its main emphasis are on scalability, performance, and interoperability.

A key element of Java EE is the Enterprise Java Beans (EJB) specification (Malena and Hapner 1999; EJB 3.0 Expert Group 2006) that describes an architecture for the execution of server-side components on application servers. EJB provides a set of services to running components that include persistence, transaction management, messaging, and load-balancing. The application server is also in charge of managing security.

Programmers define persistence and transactional properties, associated with components, declaratively by inserting annotations in the code. Persistence in Java EE has been initially (Malena and Hapner 1999) closely related to relational databases. The current version (EJB 3.0 Expert Group 2006) shows signs of convergence with the approach portrayed in JDO, moving from container-managed persistence to transparent entity-based persistence.

II.1.2.2.2 Microsoft .Net

Data-sharing in the .Net Framework (Platt 2001) has been developed along parallel lines with Java. Naturally, since it has appeared roughly five years later, it already included broader support for data-sharing, when compared with the Java initial inception. Its Base Class Library (BCL) includes support for distributed object invocation (Remoting Services), web-services, and access to relational and XML databases (ADO.NET).

Microsoft .Net ObjectSpaces (Esposito 2004) and LINQ.Net[6] (Box and Hejlsberg 2006) are two approaches aiming object-oriented persistence. ObjectSpaces is focused on providing trans-

[6].Net **L**anguage **In**tegrated **Q**uery.

parent persistence of object graphs, using a similar approach to JDO. LINQ.Net provides a set of query operators to be applied on data collections. It extends C# 3.0 with SQL-like constructs, extended to handle objects (e.g., transversal, filtering, and projection), seamlessly integrated in the syntax of the language, thus subject to metadata annotation (attributes), compile-time syntax checking, and static typing. These were previously unavailable when using SQL queries embedded in C# character strings.

The .Net approach to large-scale enterprise computing, COM+ (Löwy 2001; McKeown 2003), also adopts component-based programming. It offers a suite of services, exposed to C# via the `System.EnterpriseServices` name-space, such as resource pooling and application recycling, transactional support (relying on MTS),[7] load-balancing, messaging (relying on MSMQ),[8] event logging, and security.

COM+ services are also available to Windows applications based on native code (e.g., compiled from C++). Both ADO.Net and COM+ are based on earlier technologies, such as ADO (Microsoft 1996) and COM, DCOM (Sessions 1998) respectively, that actually pre-date the release of the .Net Framework.

SQLServer 2005 (Nunn 2005), and SQLServer 2005 Mobile that runs on mobile devices, have support for database replication and reconciliation using resolvers. Resolvers may be SQL-based stored procedures or COM components implementing a specific interface: `ICustomResolver`.

II.1.2.2.3 Other Object-oriented approaches

Gemstone (Butterwoth et al. 1991) is an early and influential commercial object-oriented database that supports C, C++ and Smalltalk. It initially included many aspects that are still present in current products. It performs caching both at the object and page level, with shadow-copying of modified pages.

It defines a query and data-manipulation language, and provides transactional support, optimistic concurrency control, and a visual schema-designer. Management of the persistent storage is performed by a stop-the-world mark-and-sweep garbage collector. Currently, GemStone Enterprise is an object-oriented distributed cache available for Java and Smalltalk integrated with the GemStone/S application server.

O2 (Lecluse et al. 1988; Deux et al. 1990; Deux et al. 1991) is another interesting early commercial effort to object databases. It has many of the features provided by GemStone. It also provides C and C++ mappings and its own query language. It includes a high-performance garbage collector (see Section III.1.3.8). Both O2 and Gemstone ensure persistence-by-reachability, i.e. all objects directly or indirectly reachable from a persistent root, and only those, are considered persistent, and thus preserved at the database.

OJB (Apache Foundation 2002), Hibernate (Iverson 2004), and its .Net port, NHibernate, are successful and widely deployed solutions to object persistence, based on object-relational mapping. Ozone (Braeutigam et al. 2002) and DB4O (db4objects, Inc. ; Paterson et al. 2006) are

[7]Microsoft Transaction Server.
[8]Microsoft Message Queueing.

two object-oriented databases used in many open-source and even commercial efforts. DB4O is very popular for small and medium-size projects because it is fully based on reflection and very easily set-up. It provides an in-memory object database, as well as object caching and replication.

Although successful to a certain extent, these systems have some shortcomings. Ozone does not support object replication nor caching at client applications. Persistent objects never leave the object server and are always invoked, by applications, via proxies. Ozone has a specification for integration with a persistence-aware garbage collection. However, the documentation does not state any actual implementation.

OJB, Hibernate and DB4O do not provide full persistence-by-reachability but allow some form of cascade-saving in which objects referenced by an object being persisted, possibly with a optional depth, are also persisted. However, objects not directly referenced or beyond a specified depth, will be neither persisted nor updated.

They also require explicit object deletion and do not enforce referential integrity, but in the simplest cases. Referenced objects may be prematurely deleted from persistence storage. Conversely, objects inaccessible from application roots may be preserved if not otherwise explicitly deleted. In practice, the programmer though in a GC-enabled environment, must perform memory management manually, w.r.t. persistence.

Summary of Chapter: In this chapter, we presented the related work concerning support for data-sharing. We addressed existing research projects and commercial technologies in the context of a proposed taxonomy. This taxonomy comprises various system-design aspects: i) system architecture, ii) programming model, iii) data-sharing model, iv) propagation of modifications, and v) portability/implementation issues. For each system-design aspect, we described its scope, and the different alternatives that can be adopted to implement it. As each of alternative was described, we made reference to relevant systems that employ it.

Additionally, we provided a study of a number of relevant research and industry systems characterizing them in the context of the proposed taxonomy. We focused on a number of early influential, and recent research projects, such as Bayou, Rover, Thor, IceCube, Mobisnap, and Javanaise. We highlighted their main features and characterized them against the taxonomy proposed earlier in the chapter.

Finally, we analyzed these projects in the context of the challenges and goals directing the work of this dissertation: i) usage of local resources, ii) support for disconnected work, iii) transparent support for object-oriented languages such as Java and C#, iv) platform portability, v) enforcement of referential integrity, and vi) adaptability. For each project and challenge/goal, when applicable, we provided details indicating full, limited or no support. We concluded with an overview of commercially available data-sharing platforms, both object-oriented and component-based, focused mainly on Java and .Net technologies.

II2 Architecture

In order to applications to function, the middleware must first make available the replicated objects they need. Desirably, the replication mechanism provided by the middleware should be efficient (concerning computing, memory and network resources), while transparent and flexible to programmers.

This chapter describes object replication in OBIWAN. It is organized as follows. First, we address the limitations of replicating full object graphs and present the advantages of incremental replication. Then, we describe how OBIWAN handles object-faults transparently, presenting the interfaces defined by its middleware core, and the OBIWAN API. Following, we provide a prototypical example of incremental replication, and describe object replication with variable depth and object clustering. We conclude with some details regarding how a number of other issues (e.g., communication, serialization, class inheritance) are addressed in OBIWAN, and explaining the various replication modes in OBIWAN.

...the case against full replication...

W.r.t efficiency, it is clearly inappropriate to offer a replication mechanism that simply downloads object repositories to clients (i.e., processes replicating objects). This is the case if object replication is performed simply by serializing all the object graph in the originating computer and send it to the destination computer. Such an approach is available when using Java (Arnold and Gosling 1996) or .Net (Platt 2001) platforms.

This approach presents some important drawbacks that we address next. It wastes memory and may even prevent application execution on clients running on mobile devices with memory restrictions. Large object graphs, if completely replicated, will exceed memory capacity in these devices. It also imposes high latency on applications. Replication of large object graphs freezes clients with slower network links. To address this, applications, or even users, must explicitly replicate an object graph in advance, which limits application flexibility. Otherwise, initial delay on application start is very high. This also limits application adaptability, and imposes reduced concurrency since applications are always believed to use complete graphs (data repositories) even if they do not access them fully. Finally, useless replication, besides wasting memory and bandwidth, it wastes user's time.

Object replication differs from replication with other programming models (e.g., files). The difference results from the fact that, with the object model, applications access data solely by navigating on the object graph, i.e., they do not access objects arbitrarily but following references. Such navigation does not occur with other programming models (e.g., when applications access plain unstructured files). Therefore, deciding how, and which objects to replicate, is strongly dependent on the navigation performed by the application. This aspect should be taken into account and, if possible, leveraged. Thus, the mobile middleware must handle incremental object replication, i.e. the partial replication of object graphs.

...the case for incremental replication...

The incremental replication of object graphs has clear advantages w.r.t. the replication of the whole reachability graph in one step. It uses memory efficiently since only the objects actually needed are replicated. This is specially tailored to mobile devices with reduced memory.

It increases network efficiency by imposing lower latency (replicating fewer objects) upon applications. Applications can invoke immediately new object replicas without waiting for the whole graph to be available. Some object replicas may be pre-fetched, with variable depth that can be optimized to suit the environment. It also has lower bandwidth costs, since replication of complete graphs happens rarely. Incremental replication also allows increased concurrency since different clients (possibly running different applications) may access different parts of object graph.

Therefore, the situations in which an application does not need to invoke every object of a graph, or the computer where the application is running has limited memory and/or network bandwidth available, are those in which incremental replication is most useful.

However, there are situations in which it may be better to replicate the whole graph; for example, if all objects are really required for the application to work, if there is enough memory, and the network connection will not be available in the future, it is better to replicate the transitive closure of the graph. The middleware must allow the application to easily make this decision at run-time, i.e. between incremental or transitive closure replication mode.

Incremental replication should be transparent to programmers. This means that they should be allowed to write code mostly free of replication concerns, besides some support API that can bootstrap the replication process. Therefore, programmers should not be forced to programmatically divide the object graph in several parts to be replicated independently. Furthermore, it should impose minimal changes to their API, to ease porting of legacy applications. In summary, object replication should not impose changes neither to the VM, nor to application code written by developers.

Thus, when an application running on a client (e.g., a mobile device) navigates on an object graph, its execution proceeds normally, navigating through the graph, as long as all objects are local. However, when it transverses an object reference, if the target object is not yet locally replicated, this generates an object-fault that must be served by the middleware.

II.2.1 Transparent Object-Fault Handling

In order to be transparent and not to impose changes to a particular VM, object-fault handling is performed by the middleware with the cooperation of extended application code (i.e., code included in application classes). Nonetheless, code to be inserted in application classes can be automatically generated to free the programmer from that task, implementing a number of interfaces defined in OBIWAN.

Broadly, object-fault handling must be able to perform three tasks:

- Detect object-fault, i.e., that an object is being invoked while it is not yet replicated.
- Resolve the object-fault. This implies obtaining a replica of the object from another process.
- Replace the proxy (as far as the calling object is concerned), with the newly obtained object replica, and resume execution. This implies invoking the intended method on the object replica.

The ideal place to detect object-faults is in proxies. Thus, proxies can serve multiple purposes: i) they prevent the serialization of the whole graph, ii) detect object fault when they are invoked for the first time, iii) instruct object replication and trigger proxy replacement (so that future invocations are direct invocation on objects and not mediated by proxies).

The creation of object replicas is delegated on the object classes themselves (since only they have access to private fields). Use of reflection on this matter limits the applicability of incremental replication (some environments may not provide reflection) and hinders performance (slows down the replication process and application execution) and imposes heavier CPU load on processes.

Proxy replacement is performed by reference patching. References to the proxy are replaced by references to the newly obtained object replica. This is instructed by the proxy invoking methods (up-calls) provided by the referencing objects.

Figure II.2.1 depicts the OBIWAN architecture (Veiga and Ferreira 2002a; Ferreira et al. 2003), portraying the same scenario presented in Chapter I.2, but with increased detail on the support for incremental replication.

II.2.1.1 OBIWAN Interfaces

Incremental object replication is performed in OBIWAN by middleware code. It implements (or is invoked by) a number of core interfaces: `IProvider`, `IDemandee`, `IDemander`, and `IProviderRemote`. Such interfaces are defined by OBIWAN and implemented by proxies and objects.[1] Interface `IProvider` is implemented by application objects and handles the creation and update of object replicas. Interface `IDemandee` is implemented by proxy-out objects and encapsulates the handling of object-faults. Interface `IDemander` is implemented by application objects and allows the replacement of references to proxies, with references to object replicas. Interface `IProviderRemote` is implemented by proxy-in objects and allows the remote invocation of methods of interface `IProvider`.

[1]By use of automatic code generation.

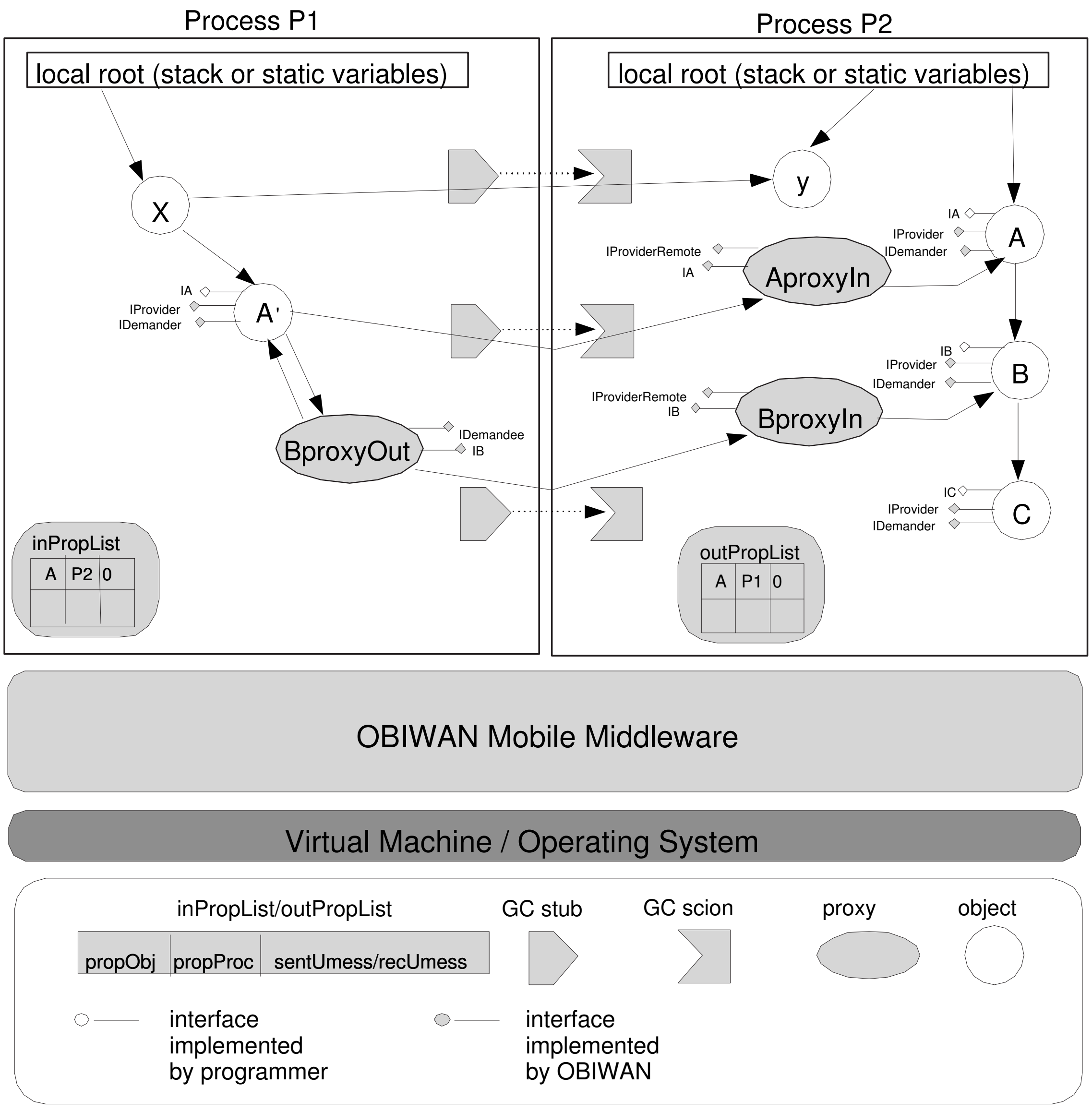


Figure II.2.1: Interfaces supporting incremental object replication in OBIWAN. Light gray interfaces in application objects are implemented by OBIWAN. Light gray objects and data structures refer to OBIWAN middleware.

For simplicity, and where there is no ambiguity, when we say that an object implements a certain interface, we obviously mean that the object is an instantiation of a class that implements that interface. These are core interfaces in the sense that programmers do not invoke them directly. In fact, they are defined to provide transparency in object replication. Programmers only need to invoke methods of the OBIWAN API (see Section II.2.2).

II.2.1.1.1 Interface `IProvider`

Interface **IProvider** (v. Table II.2.1) supports the creation and update of replicas.

Method `get` results in the creation of a replica, with internal references to other objects (handled by OBIWAN) replaced by references to proxies-out.

```
Object get(int mode, int depth );
void put(Object obj, int mode, int depth );
```

Table II.2.1: Interface `IProvider`.

Method `put` is invoked when a replica is updated with the content of another replica (e.g., when a master replica receives updates from a different process).

Arguments `mode` allows choosing different replication strategies (e.g., incremental replication, object clustering, replica refresh), while arguments `depth` allows to specify other replication depths (e.g., more than on object at a time). Both will be explained in Sections II.2.4, II.2.5, and II.2.6.9.

II.2.1.1.2 Interface `IDemandee`

```
Object demand( void );
void setProvider( IProvider prov );
void setDemander( IDemander dem );
```

Table II.2.2: Interface `IDemandee`.

Interface **IDemandee** (v. Table II.2.2) is implemented by proxy-out objects.

Method `demand` encapsulates the handling of the object-fault that occurs when the proxy is first invoked as if it was an object replica. It is responsible for actually obtaining a replica of the object it stands for. This may actually imply several actions (e.g., contacting objects in other processes, invoking middleware code, and/or triggering a specific event).

Method `setProvider` is used to associate a proxy-out object with its corresponding proxy-in object.

Method `setDemander` is used to associate a proxy-out object with the object replica that references it. Later, when the proxy is resolved, it will be notified that a replica is available.

Besides implementing interface `IDemandee`, proxies must implement the same interfaces implemented by the class they serve as proxy for. These methods, in turn, invoke method `demand`.

II.2.1.1.3 Interface `IDemander`

Interface **IDemander** (v. Table II.2.3) allows the update of references inside replicated objects. These methods are invoked when proxies are resolved.

Method `setProvider` is an "up-call" used to create (or change) the association (e.g., reference, objectID) of an object replica with another replica it may invoke either to refresh its content from it, or to update it with its own.

```
void setProvider( IProvider prov );
IProvider getProvider( void );
void updateMember( Object replica, Object member );
```

Table II.2.3: Interface `IDemander`.

Method `updateMember` is an ”up-call” used to replace references to a proxy-out object, with references to the newly created object replica. This method is invoked by proxy-out objects when an object-fault is resolved. It receives a reference to the proxy and to the new replica. It scans all fields for those that reference the proxy, and updates them so that they reference the new replica instead. After this, the newly created object replica is accessed without any indirection.

Note that there is no collision among methods `setProvider` of interfaces `IDemandee` and `IDemander`, since no object implements both these interfaces.

II.2.1.1.4 Interface `IProviderRemote`

```
Object get(int mode, int depth );
void put(Object obj, int mode, int depth );
```

Table II.2.4: Interface `IProviderRemote`.

Interface **IProviderRemote** inherits from IProvider so that its methods can be invoked remotely, i.e., from a different process. It hides from interface `IProvider` any details regarding distribution.

II.2.1.1.5 Application-code Interfaces

Interfaces **IA**, **IB**, and **IC** are the public interfaces of objects A, B and C, respectively, designed by the programmer. They define the methods that can be invoked on these objects. [2] These interfaces must be also implemented by proxy-out and proxy-in objects for classes A,B, and C.

II.2.2 OBIWAN Application Programming Interface for Replication

As already mentioned, application programmers never need to invoke OBIWAN interfaces explicitly. The OBIWAN library provides a number of overloaded class (*static*) methods that

[2]The same reasoning applies to objects X and Y; but given that they are not involved in the replication scenario, we do not consider them.

```
Object GetObject( string objName, int mode, int depth );
void PutObject( string objName, Object obj, int mode, int depth );
```

Table II.2.5: OBIWAN Application Programming Interface for Replication.

allow application programmers to replicate and update object graphs, based on i) object addresses (e.g., URLs, that are presented in Table II.2.5), ii) objects already replicated, iii) proxies, or iv) lookup on a name service.

These methods will, in turn, also invoke corresponding methods `get` and `put` on corresponding proxy-in objects. There are other overloaded methods to account for most common cases (e.g., to allow omitting certain parameters).

II.2.3 Prototypical Example of Incremental Replication

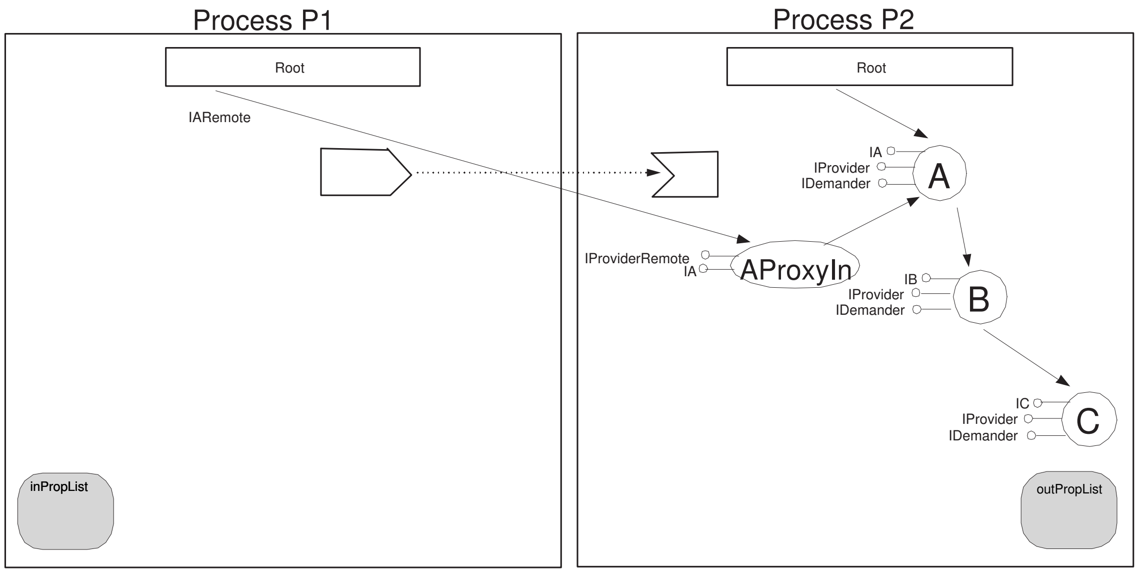


Figure II.2.2: (a) Incremental Replication: Initial situation .

II.2.3.1 Initial Situation

Taking into account the architecture presented in Figure II.2.1, we now describe a prototypical example that illustrates the incremental replication of an object graph from process P2 to process P1, the occurring object-faults, and their corresponding resolution.

The initial situation (a), portrayed in Figure II.2.2 is the following: i) P2 holds a graph of objects A, B and C; ii) object A is registered in a name server, which returns a proxy-in to it (AProxyIn) that can be invoked remotely, and iii) P1 holds a remote reference to object AProxyIn, that was obtained from a name server, since object A is the root of an object graph.

Objects A, B and C in P2 were created by the programmer or loaded from persistent store. Their replicas (A′, and later B′, and C′) are created upon request originated in P1 (either explicitly by the programmer or triggered by an object-fault). All other objects, i.e. proxies-in and proxies-out are part of the OBIWAN platform and are transparent to the programmer, except AProxyIn.

Figures II.2.2 and following show, for each process, the content of data structures regarding replication (i.e., `InPropList` and `OutPropList`). For each object, the interfaces implemented by it are also shown.

This example starts with the application running in P1 requesting the creation of replica A′. This may be performed resorting to method `GetObject` with the remote reference (to AProxyIn). This will in turn invoke AProxyIn.get remotely. Initially, object replication must always start with this bootstrap phase, when there are no previously replicated objects.

II.2.3.2 Incremental Replication of Object A

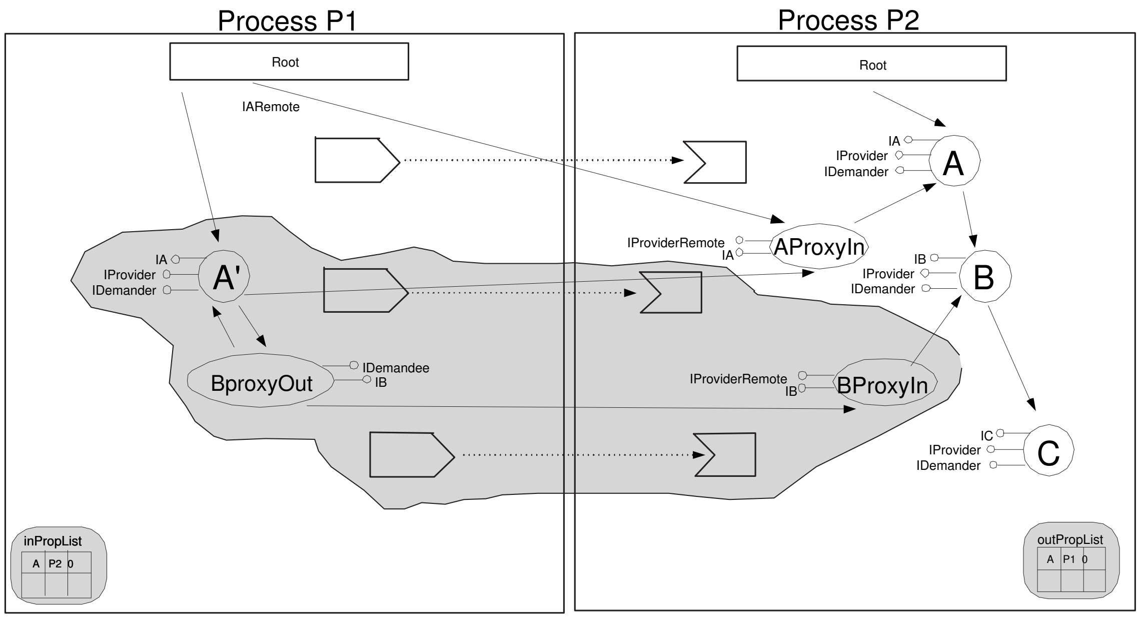


Figure II.2.3: (b) Incremental Replication of object A. Objects created in this step are shaded.

Upon invocation of `AproxyIn.get` in P2, this method will result in the execution of the following steps:

1. invoke method A.get
2. method A.get starts by creating A′ in P2, and copying all data of primitive fields from A into A′;

For each reference A holds (only to B in this case), perform the following steps:

3. create the corresponding ProxyIn objects (only BProxyIn in this case) in P2; in the constructor of BProxyIn, set an internal reference pointing to B;

4. create a ProxyOut object for each ProxyIn created in the previous step (only BProxyOut in this case) in P2;
5. set the internal reference of A′ (of type IB) so that it points to BProxyOut;
6. invoke BProxyOut.setProvider(BProxyIn) so that BProxyOut points to BProxyIn;
7. invoke BProxyOut.setDemander(A′) so that BProxyOut also points to A′;

After every reference in A has been handled as described in the previous steps:

8. create (since it does not exist yet) an entry for A in the OutPropList of process P2. This entry accounts for the fact that object A is being replicated to process P1.
9. method A.get returns the new objet replica A′.
10. method AProxyIn.get terminates simply by also returning A′.
11. as a result, A′ and BProxyOut are automatically serialized by the underlying virtual machine and sent to P1.
12. when replica A′ is instantiated in P1, an entry for A is created (since it does not exist yet) in the InPropList of process P1. This entry accounts for the fact that object A has been replicated from process P2.

This results in situation (b) in Figure II.2.3 where the just created objects are shaded. Proxy-in objects never leave the process where they are created; they are not serialized. This prevents the whole graph from being replicated to process P1. Thus, proxy-out objects act as the frontier of the graph already replicated.

II.2.3.3 Object-Fault Detection and Incremental Replication of Object B

Later, the code in A′ may invoke any method, which is part of the interface IB (exported by B), using its internal reference. For simplicity, we will call this method `m`. As a result, it will actually invoke method (e.g., `BProxyOut.m`) on BProxyOut (that A′ sees as being B′). For transparency, this requires that the system supports an "object-fault" mechanism. All the methods of interface IB in BProxyOut, follow a similar behavior: they i) invoke method BProxyOut.demande to handle the object-fault, ii) then invoke the intended method on the object replica just obtained (e.g., B′.m), and iii) finally returning the result to the calling object (that will not be aware that B′ did not exist yet).

The detailed steps for handling the object-fault regarding object B, are the following (object invocations and their outcome, such as creation of objects, references, and updates to structures regarding replication, are presented with step number, in Figure II.2.4):

1. method BProxyOut.m starts by invoking its demand method BProxyOut.demand

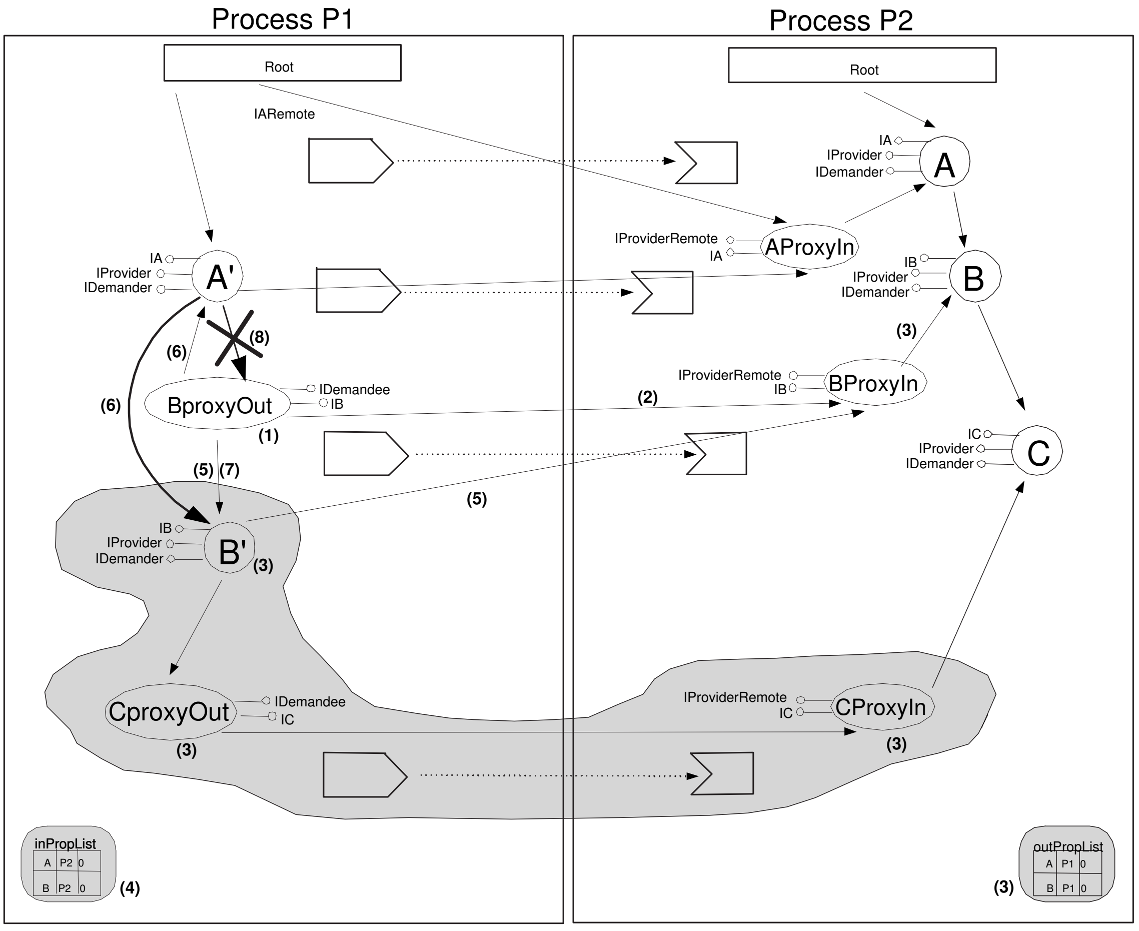


Figure II.2.4: (c) Incremental replication of Object B showing intermediate step of replacement of BProxyOut. Objects created on this step are shaded.

Since object B is not yet replicated (there is no entry regarding B in the InPropList of P1):

2. BProxyOut invokes method BProxyIn.get (BProxyIn is BProxyOut's provider), to obtain a replica of object B, that will proceed in a similar way as explained previously for object A.

3. BProxyIn.get invokes B.get that will proceed in a similar way as described prior for A.get: creates B′, CProxyOut, CProxyIn and sets the references between them; once this method terminates, as illustrated in Figure II.2.4, B′, BProxyOut and CProxyOut are all in P1, CProxyIn is in P2, and BProxyOut points to B′; note that A′ and BProxyOut still point to each other; an entry for B is created in the OutPropList of process P2, accounting for the fact that B is being replicated to process P1.

4. when replica B′ is instantiated in P1, an entry regarding B is created (since it does not exist yet) in the InPropList of process P1. This entry accounts for the fact that object B has been replicated from process P2.

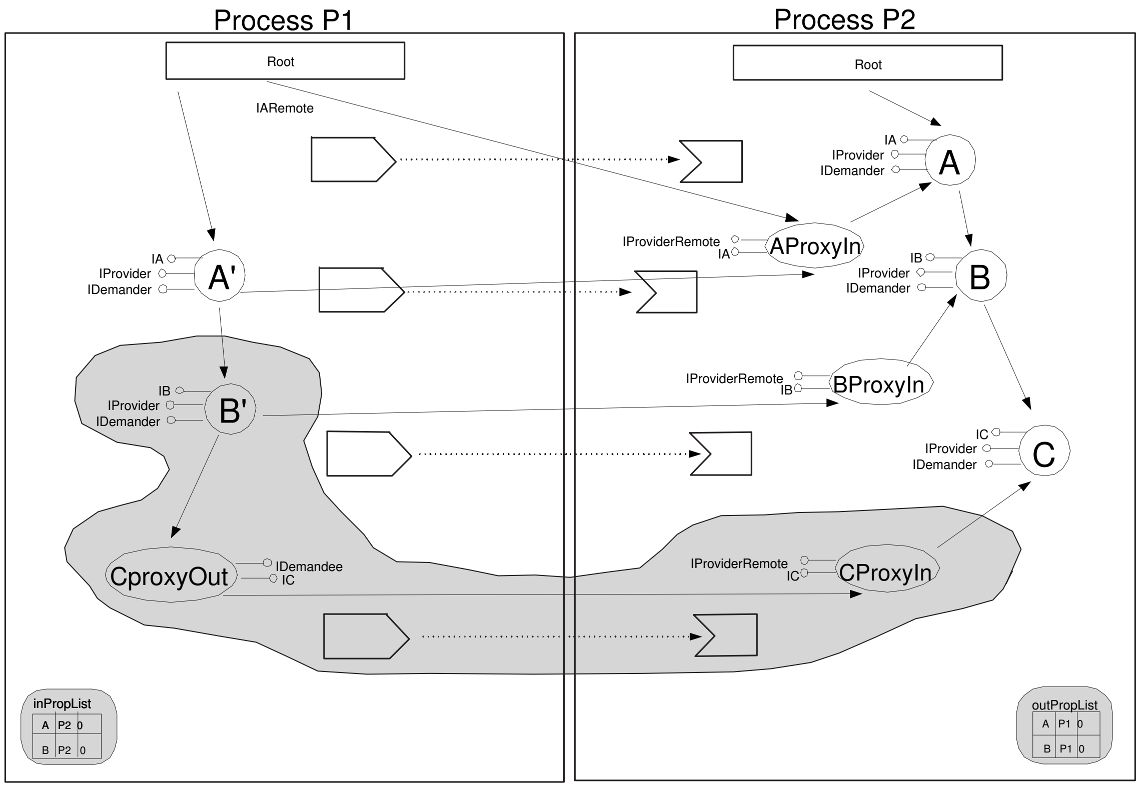


Figure II.2.5: (d) Final situation with object B replicated and BProxyOut discarded. Objects created on this step are shaded.

II.2.3.4 Replacement of BProxyOut

Once object B is replicated from P2 into P1, the application can resume execution. This requires that B′ is invoked with the same method (recall method `m`), as originally intended, that was invoked on BProxyOut. Furthermore, since object replica B′ is already available, future method invocations from A′ should not have to be mediated by BProxyOut, that should be discarded. Thus, A′ must now reference B′ instead of BProxyOut. Replacing BProxyOut with B′, and resuming execution is performed by the following steps:

5. BProxyOut invokes B′.setProvider(this.provider) so that B′ also points to BProxyIn; this is needed because the application can decide to update the master replica B, by invoking method B′.put that in turn will invoke BProxyIn.put, or to refresh replica B′ (method BProxyIn.get);

6. BProxyOut invokes A′.updateMember(B′,this) so that A′ replaces its reference to BProxyOut with a reference to B′;

7. finally, BProxyOut invokes the same method on B′ that was invoked initially by A′ (that triggered this whole process) and returns accordingly to the application code;

8. from this moment on, BProxyOut is no longer reachable in P1 and will be reclaimed by the garbage collector of the underlying virtual machine.

Figure II.2.5 illustrates situation (d) after BProxyOut has been reclaimed by the garbage collector of the underlying virtual machine. It's important to note that, after situation (d), further invocations from A′ on B′ will be normal direct invocations with no indirection at all. Later, when B′ invokes a method on CProxyOut (standing in for C′ that is not yet replicated in P1) an object fault occurs and will be solved with a set of steps similar to those previously described.

II.2.4 Incremental Replication with Variable Depth

The replication mechanism just described is very flexible in the sense that allows each object to be individually replicated. However, this has a cost associated with the latency imposed by having to send a replication request over the network for each object being replicated.

This could be avoided since it is expected that other objects referenced by the one being replicated will also be accessed in the near future. Therefore, it would be advantageous to replicate part of the graph, i.e., those objects that are indirectly referenced (to a certain depth) from the object whose fault is being handled. Thus, it should be possible to perform incremental replication of more than one object at a time.

To address this, OBIWAN allows an application to replicate objects using variable replication depths larger than one (e.g., 25, 100). This is an intermediate solution between: i) having the possibility of incrementally replicate each object, or ii) replicating the whole graph. This is achieved by invoking method get of interface IProvider recursively, until the specified depth is fulfilled. This way, an object creates a replica of itself and instructs all objects referenced by it, to do the same. The result is a partial copy of the object graph, up to a certain depth. Objects at the specified depth are not replicated. In their place, a proxy-out object is created.

Obviously, if there are not enough objects to satisfy a certain replication depth, incremental replication will behave as replication of the complete transitive closure, i.e., it will replicate all reachable objects.

The actual value used for depth parameter may be explicitly provided by the programmer or delegated to OBIWAN. OBIWAN decision is ruled by the policies loaded (possibly taking the execution context into account) in the system and is addressed in detail in Part IV. Thus, incremental replication in OBIWAN is highly dynamic.

When replicating objects with depth larger than one, some replication steps can be omitted. Without loss of generality, in the previous example, the creation of BProxyOut would not be needed (thus, it would not be performed). This is because replicas A′ and B′ would be created in the same replication request and transferred (together with CProxyOut) to process P1. This is because ProxyOut objects are only created when the replication depth reaches zero.

W.r.t proxy-in objects, there must be created specific proxies for each of the objects being replicated (e.g., AProxyIn, BProxyIn, CProxyIn in P2). The object replicas created and transferred to P1 also carry references (i.e, remote references) to these proxies-in. These references allow direct individual access to remote objects A,B, and C from process P1 (e.g., for remote invocation, content refresh, or direct update).

Registration of replicas of intermediary objects in InPropList must be performed when these objects are replicated into process P1. This may be performed in several ways that will be made clear in Sections II.2.6.4 and II.3.3.4.

Incremental replication with variable depths reduces application latency and the number of times it must access the network, while profiting from available bandwidth.

II.2.5 Object Clustering

In spite of the advantages provided by incremental replication, replicating objects with depths larger than one still creates individual proxy-in objects for each of the objects being replicated.

There is a cost associated with the creation of such individual proxies since extra objects are being created. This may be unnecessary since an application might not need neither to access these objects remotely, nor update them individually. To minimize this cost, OBIWAN allows an application to replicate and regard a set of objects as a whole, i.e., a cluster. This is achieved using a predefined value in argument `mode` of method `get` of interface `IProvider`.

A cluster is a set of objects that are part of a reachability graph, with a single pair of proxy-out/proxy-in objects. For example, if an application holds a list of 1000 objects, it is possible to replicate a part of the list so that only 100 objects are replicated and a single pair of proxy-in/proxy-out is effectively created and transferred between processes. This makes replication faster but no longer allows neither refresh, nor update of individual replicated objects transparently. It also disables direct access to remote objects.

As with incremental replication without clustering (with variable replication depth), the depth of each cluster of objects may be dynamically defined.

In summary, OBIWAN provides the following alternatives to object replication:

- object-fault handling, i.e., replicate objects as needed, one at-a-time.
- incremental replication with variable depths.
- incremental replication with object clustering, also with variable depths.
- full-graph replication, i.e., transitive-closure mode.

In Chapter II.4 we present the performance results for relevant scenarios.

II.2.6 Other Issues

There are a number of aspects that must be taken into account when implementing incremental replication on top of a specific virtual machine. This section offers an overview of some important architectural aspects and the issues they raise, leaving the details of how they are addressed in the context of each implementation, to the next chapter. These aspects include:

1. communication between processes, w.r.t. the availability of a remote method invocation mechanism.
2. serialization mechanism provided by the virtual machine, w.r.t. transference of object replicas and proxies-out.
3. object identity, w.r.t. ensuring that application code sees only one (and always the same) replica of each object, as it would happen in a non-replicated environment, where an object is embodied by a single instance.
4. object instantiation, w.r.t. detecting when replicas are received in a process, in order to update data structures regarding replication.
5. public class fields, w.r.t. providing transparency of proxy objects.
6. supporting inheritance, w.r.t. performing incremental replication of fields belonging to classes in different levels of a class hierarchy.
7. existing libraries, w.r.t. handling objects of classes not extended by OBIWAN.
8. detection of object modification, w.r.t. providing information regarding modified objects to other modules implementing additional functionality (e.g., transactions).
9. replication modes, w.r.t. how to instruct methods of interface `IProvider` to perform in different contexts (e.g., incremental replication vs object clustering, execution in the client vs server).

II.2.6.1 Communication Between Processes

Invoking `IProviderRemote` objects located in a different process must ultimately rely on some form of distributed communication among processes.

If this service is provided by the underlying virtual machine, in the form a remote method invocation mechanism (i.e., invoking individual objects remotely), it may be used directly. If just a remote procedure call mechanism is available (i.e., it is not possible to invoke individual objects), it must be adapted. This will be addressed in the next chapter. If none is available, an alternative mechanism must be used instead.

Thus, different implementations of OBIWAN may implement access to `IProviderRemote` objects (proxies-in) differently. Nonetheless, the fundamental architecture decisions remain unchanged.

II.2.6.2 Serialization

We assume that, when some form of RPC or RMI mechanism is available, it is also accompanied by a form of serialization (even if limited). This is needed in order to be able to transfer the content of replicated objects, namely when more than one object is being replicated at-a-time.

Virtual machines for desktop computers provide full object serialization while virtual machines for mobile devices (e.g., Java Micro Edition and .Net Compact Framework) do not. In the next chapter, we describe how OBIWAN can be implemented in both these scenarios.

II.2.6.3 Object Identity

Obviously, there must be a way to determine if two object replicas refer to the same object. Thus, objects must have a globally unique ID given to them when they are involved in replication for the first time (either being replicated, or proxies to them being created). This field associates different replicas of the same object (e.g., when replicated objects in P1 are refreshed with more recent content available at P2, or when objects in P2 are updated with the content of replicas that were modified in P1). Furthermore, this global unique ID is helpful in determining if different proxies stand in for the same object.

II.2.6.4 Object Instantiation

When objects are replicated from and to a process for the first time, it is necessary to register this fact in the corresponding entries in the OutPropList and InPropList, respectively.

When object replicas are created, during execution of method get of interface IProvider, it is possible to test if the object has already been replicated and, if not, create and insert the corresponding entry in the OutPropList.

When object replicas arrive to (or are received in) a process, it is necessary to verify if the object has been replicated before. If it has not, an entry referring to it is created in the InPropList of the process. This can be performed when objects are instantiated during de-serialization, if this mechanism is customizable. If not, it must be performed explicitly when objects are instantiated.

The different ways to achieve this with automatically generated code are discussed in the next Chapter.

II.2.6.5 Public Class Fields

The use of proxies-out raises an architectural issue w.r.t. class design. Since both proxies-out and objects share an interface but not an implementation (e.g., B′ and the corresponding BProxyOut share IB but have different implementations) objects can only be manipulated by means of method invocation (i.e. no direct access to internal data). Thus, classes should not have public fields that could be accessed by other objects, since these will not be available in proxies-out that, by definition, do not carry object data.

In the case of Figures II.2.1 to II.2.5, this means that the code written by the programmer in A′ can not access directly internal data of B′. This is due to the fact that B′ may not exist yet; instead, there is BProxyOut in P1 that implements IB but in such a way that it detects the fault of B′; then, BProxyOut will create B′ so that the invocation can proceed normally.

We find this not to be an important restriction. As a matter of fact, this is a sensible way of manipulating objects, only through methods, thus ensuring encapsulation. Note that this aspect is also present, for example, in Microsoft ActiveX components (Microsoft 1996) and Java Beans (Englander and Loukides 1997).

Additionally, this can be circumvented in languages that support property fields in their syntax (e.g., C# or VB.Net). These languages allow the definition of public methods (set and get) that are used by the programmer with the same syntax as public object fields.

II.2.6.6 Supporting Inheritance

OBIWAN architecture for incremental object replication does not force programmers to derive their classes from a specific top-level class that replaces the base object class of the virtual machine. Programmers are free to use inheritance while developing/writing classes, since OBIWAN solely mandates that the classes implement a set of interfaces.

Furthermore, OBIWAN architecture allows the programmers to define class hierarchies so that objects are incrementally replicated, where the private fields of superclasses are also incrementally replicated (not just the fields of the subclass). OBIWAN ensures that only one replica is actually created for each object, regardless of the number of superclasses it inherits from. OBIWAN also ensures proxy replacement at superclasses. This support should not, and needs not, rely on runtime reflective capabilities to circumvent field protection. OBIWAN code in subclasses does not use reflection to access private fields of superclasses (more details in Section II.3.3.6).

II.2.6.7 Existing Libraries

To be supported by OBIWAN, w.r.t. incremental replication, classes must have been previously extended as mentioned in Chapter I.2. Thus, classes belonging to (unextended) existing libraries are regarded, by default, simply as private data of other objects recognized by OBIWAN.

II.2.6.8 Detection of Object Modification

OBIWAN supports the update of partial or full graphs of objects. However, to some applications or higher level components, it may be useful to flag modified objects individually. Programmers may explicit inform OBIWAN that an object has been modified, or provide hints to OBIWAN of which methods may potentially modify object state.

This information can be used by transactional support in order to send to another process only the objects that were actually modified. These can be copied (with their references treated, i.e., replaced by proxies if they refer to objects not included in the set of object being updated), inserted in a single structure (e.g., an array) and sent to the process where the updates should be committed.

II.2.6.9 Replication Modes

OBIWAN defines a number of flag constants to characterize replication modes (recall the first argument of methods of interface `IProvider` and its extension `IProviderRemote`). Some of them may be also used by programmers when invoking OBIWAN API methods `GetObject` and `PutObject`. Several modes (if not incompatible) can be combined within the same replication operation. The supported modes are:

- `TRANSITIVE_CLOSURE`: This replication mode instructs OBIWAN to replicate complete object graphs, i.e., not incrementally.
- `INCREMENTAL_REPLICATION`: This is the default replication mode in OBIWAN. Incremental object replication uses the depth specified explicitly by the programmer or provided by an OBIWAN property. In incremental replication mode, objects are only replicated once for each pair of processes. If during the replication of a partial graph, OBIWAN reaches an object that was already replicated, it simply returns a proxy for it, since it is already available at the client process.
- `CLUSTERING`: In this replication mode, OBIWAN behaves as in INCREMENTAL_REPLICATION mode, with the exception that only one proxy-in object is created for each cluster that is replicated. When a cluster is replicated, the objects beyond the specified depth are replaced by proxies-out. These proxies-out reference proxy-in counterparts that serve as heads to other clusters. Cluster depth may also be explicitly defined or left to OBIWAN to decide.
- `CLIENT_MODE`: This replication mode informs methods of interface IProvider that they are being invoked in the context of one process that triggered object replication (or is sending updates).
- `SERVER_MODE`: Complementary to CLIENT_MODE, this replication mode informs methods of interface IProvider that they are being invoked in the context of the process that is being requested to perform object replication or to receive updates. This and the previous mode also instruct OBIWAN which data structures to manipulate, w.r.t. regarding replication (i.e., InPropList and OutPropLists).
- `REFRESH`: This replication mode instructs OBIWAN that the objects being replicated will refresh (i.e., update the contents of) object replicas already available at the requesting process. Therefore, the default behavior associated with INCREMENTAL_REPLICATION and CLUSTERING must be overridden w.r.t. avoiding the replication of the same object twice between the same pair of processes. To achieve this, the object replicas are created in the process receiving the request (via execution of method get of interface IProvider) and, once received at the requesting process, are used to update the content of the already existing objects (via execution of method put of interface IProvider). Thus, refreshing of object replicas requires a sequence of two operations: first, that object replicas be created (method get) in one process and, then, applied as updates (method put) in the other.
- `UPDATE`: Conversely to mode REFRESH, this mode instructs OBIWAN that the replicated objects available at the requesting process will be transferred to another process in order

to update its replicas (e.g., a master replica). Updating replicas at another process requires that object replicas are created (method get) in the requesting process and applied as updates (method put) in the receiving process.

- `RE_USE_OBJECT`: This replication mode is used to support incremental object replication in the presence of class inheritance. It instructs method get invoked in each superclass not to create additional replicas of the object being replicated. It should reuse a replica already created during the same replication operation, and update its fields concerning the superclass.

Since programmers only invoke directly methods of the OBIWAN API (`GetObject` and `PutObject`), and not methods of interface `IProvider`, API methods must sequence the execution of methods `get` and `put` of interface `IProvider`, with the appropriate parameters, according to each situation (i.e., replicating objects incrementally, refreshing object replicas, or sending updates).

Table II.2.6 describes, for each situation, the sequence of invocations that takes place at the client and server. The default replication mode is assumed to be `INCREMENTAL_REPLICATION` and therefore can be omitted. Details and parameters regarding object clustering, replication depth, and inheritance are omitted since they do not entail changes to the invocation sequences presented. Proxy indirection is also omitted as it is implied when execution flow moves between client and server.

Situation	Client	Server
Incremental Object Replication	`GetObject(name,mode,depth);` (*returned objects are instantiated at client*) (*execution continues*)	`get(SERVER_MODE | mode,depth);` (*replicated subgraph is returned to client*)

Refreshing Object Replicas	`GetObject(...,REFRESH,...);`
	 (*returned objects are instantiated at client*)
	 (*refresh local replicas at client invoking...*)
	 `put(CLIENT_MODE|REFRESH,...);`
	 (*execution continues*)
		`get(SERVER_MODE|REFRESH,...);`
		 (*replicated subgraph is returned to client*)

Sending Updated Replicas	`PutObject(...,UPDATE,...);`
	 `get(CLIENT_MODE|UPDATE,...);`
	 (*replicas of updated objects sent to server*)
	 (*execution continues*)
		(*updated replicas instantiated at server*)
		 (*update objects at the server invoking...*)
		 `put(SERVER_MODE|UPDATE,...);`

Table II.2.6: OBIWAN API methods and corresponding sequences of invocations of `IProvider` methods.

Summary of Chapter: In this chapter, we presented in detail the architecture of transparent object-fault handling, and incremental object replication in OBIWAN. The architecture is based on a set of OBIWAN core interfaces which enable these mechanisms: i) `IProvider` related to the creation and update of object replicas, ii) `IDemandee` w.r.t. object-fault handling by proxy objects, and iii) `IDemander` allowing proxy replacement, once the corresponding object is replicated. These core interfaces are implemented by middleware code in proxies-out, proxies-in, and application objects, which is automatically generated. The programmer is oblivious of them. To ensure transparency to the programmer, proxy-out objects must also implement the same interfaces as application objects. Invocation of a proxy-out triggers object replication. Additionally, we presented the OBIWAN API exposed to programmers.

We provided a detailed illustrated prototypical example showing how object-fault detection, and incremental replication work, portraying the sequences of steps involving the creation of object replicas, proxies, and pointing out invocations of methods described in the OBIWAN interfaces. We further described object replication with variable depth and object clustering. The mechanisms described are portable and can be implemented in widely deployed virtual machines, such as Java and .Net.

We also addressed some relevant issues, such as support for communication between processes and object serialization, which are leveraged by OBIWAN if they are provided by the underlying VM; otherwise, they must be addressed by the OBIWAN implementation. Support for class inheritance without resort to reflective capabilities during runtime is also motivated. Finally, we described the various replication modes in OBIWAN, and how the OBIWAN API orchestrates the execution of OBIWAN core interfaces.

II3 Implementation

Incremental object replication in the the OBIWAN architecture has been implemented and tested in a number of prototypes, and example applications. Figure II.3.1 depicts an overview of the prototype implementations of OBIWAN, w.r.t. the specific issues addressed, and functionality provided by each one. These prototypes can be divided in two groups, depending on the support they provide for mobile constrained devices (e.g., PDAs).

Two prototypes: i) OBIWAN.Java (Veiga and Ferreira 2002a; Ferreira et al. 2003), and ii) OBIWAN.Net (Veiga and Ferreira 2002b) target exclusively laptop and desktop computers, running on top of the Java Virtual Machine, and .Net Common Language Runtime, respectively.

The remaining prototypes M-OBIWAN, OBI-Web, OBI-Per (Veiga et al. 2004; Santos et al. 2004) all build on top of the OBWIAN.Net prototype. The M-OBIWAN prototype enables mobile constrained devices to run client processes, in the sense that they can replicate objects from other processes, and send updated objects to them later. To serve object requests, mobile constrained devices must have a Mobile Web Server (Nicoloudis and Pratistha 2003) running.

Support for serving object graphs directly from a web-based application server is addressed in the OBI-Web prototype, while persistence for object repositories is described in OBI-Per. OBI-VS is a plug-in to integrate OBIWAN.Net, and related prototypes, with the Visual Studio development environment.

The prototypes implemented do not require any modification of either JVM or CLR internals. This fact is key for OBIWAN portability. Figure II.3.2 describes how the prototype implementations of OBIWAN are integrated, and how each one communicates with instances of

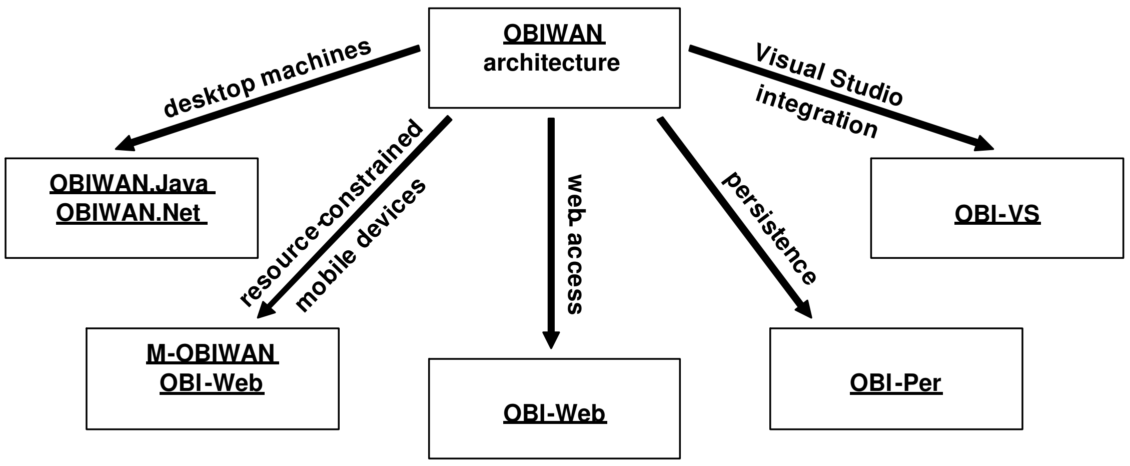


Figure II.3.1: Overview of OBIWAN implementations.

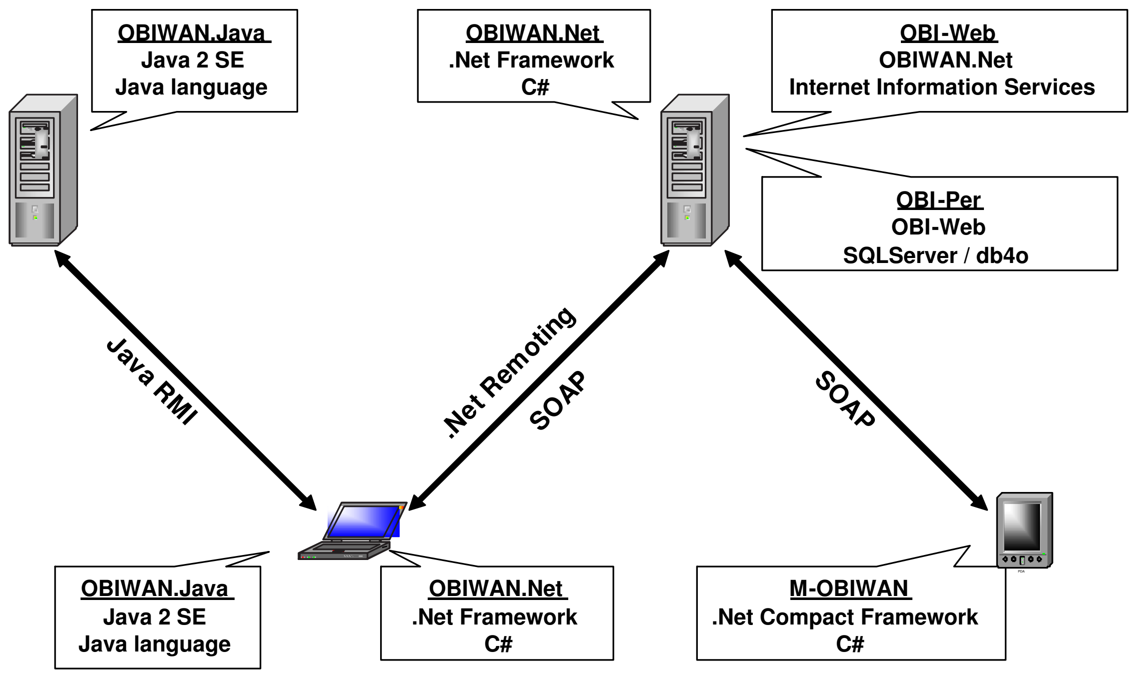


Figure II.3.2: Integration of OBIWAN prototype implementations. Each prototype indicates other prototypes, and external software, it relies on.

identical and different prototypes.

Communication between processes is performed resorting to Java RMI (Remote Method Invocation), .Net Remoting Services, and SOAP (Simple Object Access Protocol). The reason for using SOAP is twofold: i) it allows platform inter-operability between Java and .Net implementations; ii) it also serves as the only option of communication for implementations on mobile constrained devices, as they have very limited communication services.

The next two sections are dedicated to the description of the OBIWAN desktop prototypes (OBIWAN.Java and OBIWAN.Net) and M-OBIWAN, respectively. Section II.3.3 thoroughly describes the implementation aspects common to all the prototypes, i.e., the core of incremental object replication in OBIWAN. The remaining sections of the chapter briefly present details and extensions specific to the other prototypes (OBI-Web, OBI-Per), and support for integrated development environments.

II.3.1 OBIWAN Desktop Implementations

The differences between OBIWAN.Java and OBIWAN.Net are not extensive, since the two underlying virtual machines offer similar characteristics. Thus, the next subsections will describe only the implementation details that are not common, and highlight the differences between the two prototypes.

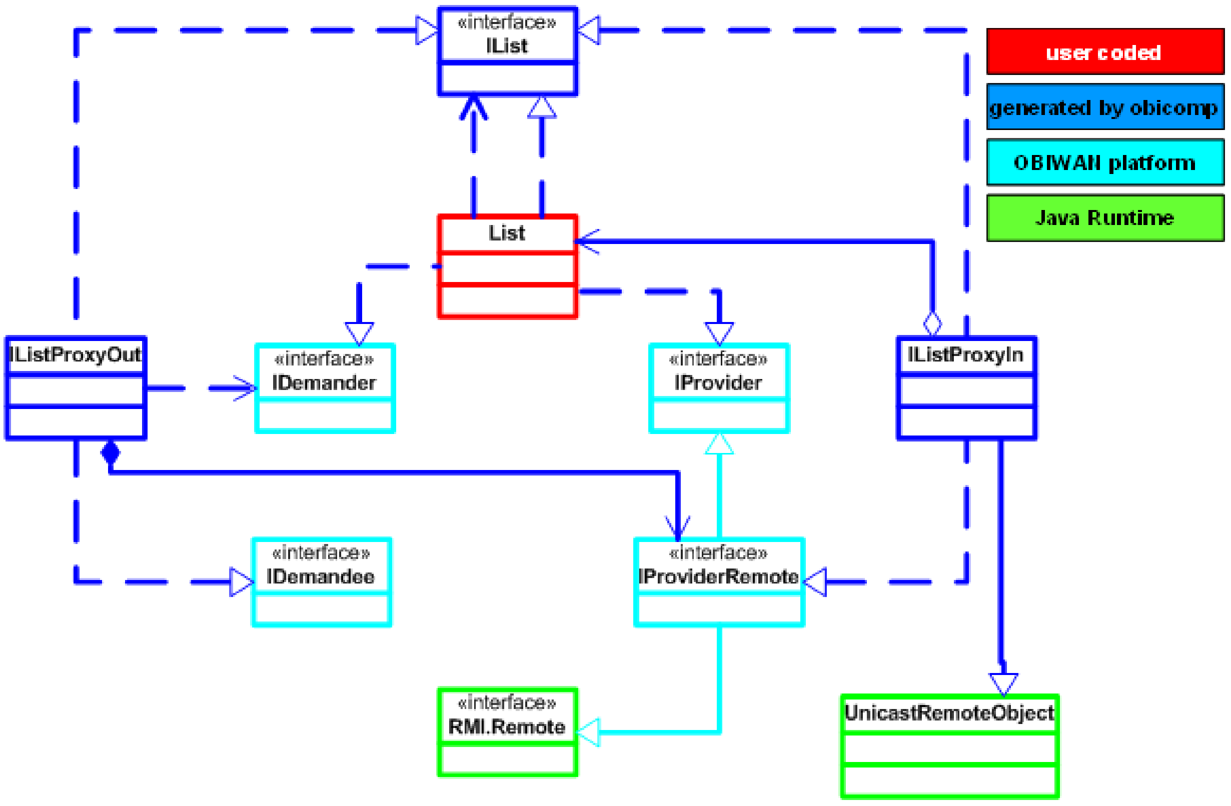


Figure II.3.3: UML class diagram for OBIWAN.Java implementation.

II.3.1.1 OBIWAN.Java

The OBIWAN.Java prototype runs on top of Java 2 Standard Edition (version 1.3 and above). It includes the implementation in Java of: i) the OBIWAN interfaces (`IProvider`, `IDemander`, `IDemandee`), ii) `OBIRep` data structures, iii) OBIWAN utility class (with API methods and other helper functions), and iv) `obicomp.java` compiler. Thus, OBIWAN.Java is provided a set of `.class` files, and a shell script/batch file.

The diagram in Figure II.3.3 shows the result of class extension, by `obicomp.java`, on an example class (`List`). OBIWAN interfaces are constant, so they do not have to be generated every time.

Communication between processes is performed using Java Remote Method Invocation (RMI) and Java Serialization. Thus, OBIWAN interface `IProviderRemote` extends interface `java.rmi.Remote`. Proxies-in belong to class `IListProxyIn`, and extend class `java.rmi.UnicastRemoteObject`, which is a base class for remote objects in Java. Classes `List` and `IListProxyOut` must implement interface `java.io.Serializable`.

Class `List` must extend the Java Serialization mechanism in order to allow registering objects in the `OBIRep` structures, when the objects are replicated into a process, for the first time. This is achieved, via customization of Java Serialization, by implementing methods `writeObject` and `readObject`. Implementation of method `writeObject` simply delegates to method `defaultWriteObject` of the `ObjectOutputStream` provided as argument. Method `readObject` delegates to method `defaultReadObject` of the `ObjectInputStream` provided as argument and then, checks the `OBIRep` of the process with the object's globally unique ID (hereafter referred simply as objectID).

The `obicomp.java` compiler is a shell script (or batch file) that invokes a number of java classes in sequence (`OBICreateInterface`, `OBIProxyGenerator`, and `OBIParser`). It initially scans the file in search of a specific comment (`/*[OBIWAN]*/`) prepended before either a class or field name. This informs `obicomp.java` that the class should be extended and which fields should be replicated incrementally. After the class is compiled, it is analyzed using Java reflection mechanisms, w.r.t. inheritance, field types, and public methods.

Generation of proxy classes is performed by an autonomous class that creates the source files and writes code that implements the patterns described. After this, the proxy classes are compiled using `javac`. The proxy-in class is also fed to `rmic`, the Java RMI compiler[1].

Extension of application class (e.g., `List`) is performed by source code augmentation, i.e., by a class that inserts into and modifies text in the class source file, after it is saved in a backup. All code generated by `obicomp.java` is enclosed in special comments to ease debug. After this the extended class can be recompiled and the extension process is finished.

II.3.1.2 OBIWAN.Net

The OBIWAN.Net prototype runs on top of .Net Framework (version 1.0 and above). It includes the implementation in C# of: i) the OBIWAN interfaces (`IProvider`, `IDemander`, `IDemandee`), ii) `OBIRep` data structures, iii) OBIWAN utility class (with API methods and other helper functions), and iv) `obicomp.net` compiler. Thus, OBIWAN.Net is provided as an assembly, and a shell script/batch file.

The diagram in Figure II.3.4 shows the result of class extension, by `obicomp.net`, using the same example class (`List`). The main differences from OBIWAN.Java are highlighted. OBIWAN interfaces are constant, so they do not have to be generated every time.

Communication between processes is performed using .Net Remoting Services, and .Net Serialization (with Binary formatter). OBIWAN interface `IProviderRemote` simply extends interface `IProvider`.[2] Proxies-in belong to class `IListProxyIn`, and extend class `System.MarshalByRefObject`, which is a base class for remote objects in .Net. Classes `List` and `IListProxyOut` must be tagged with attribute `[Serializable]`.

Class `List` extends the .Net Serialization mechanism,to allow registering objects in the `OBIRep` structures. This is achieved, via customization of .Net Serialization, by implementing interface `IDeserializationCallback` with method `OnDeserialization`. It simply checks the `OBIRep` of the process with the objectID of the object being replicated.

The `obicomp.net` compiler, similarly to `obicomp.java`, is a batch file that invokes a number of .Net programs in sequence. After the class is compiled, it is analyzed using .Net

[1]This final step is no longer needed in recent versions of Java.
[2]There is no need to tag interfaces explicitly as remote in .Net.

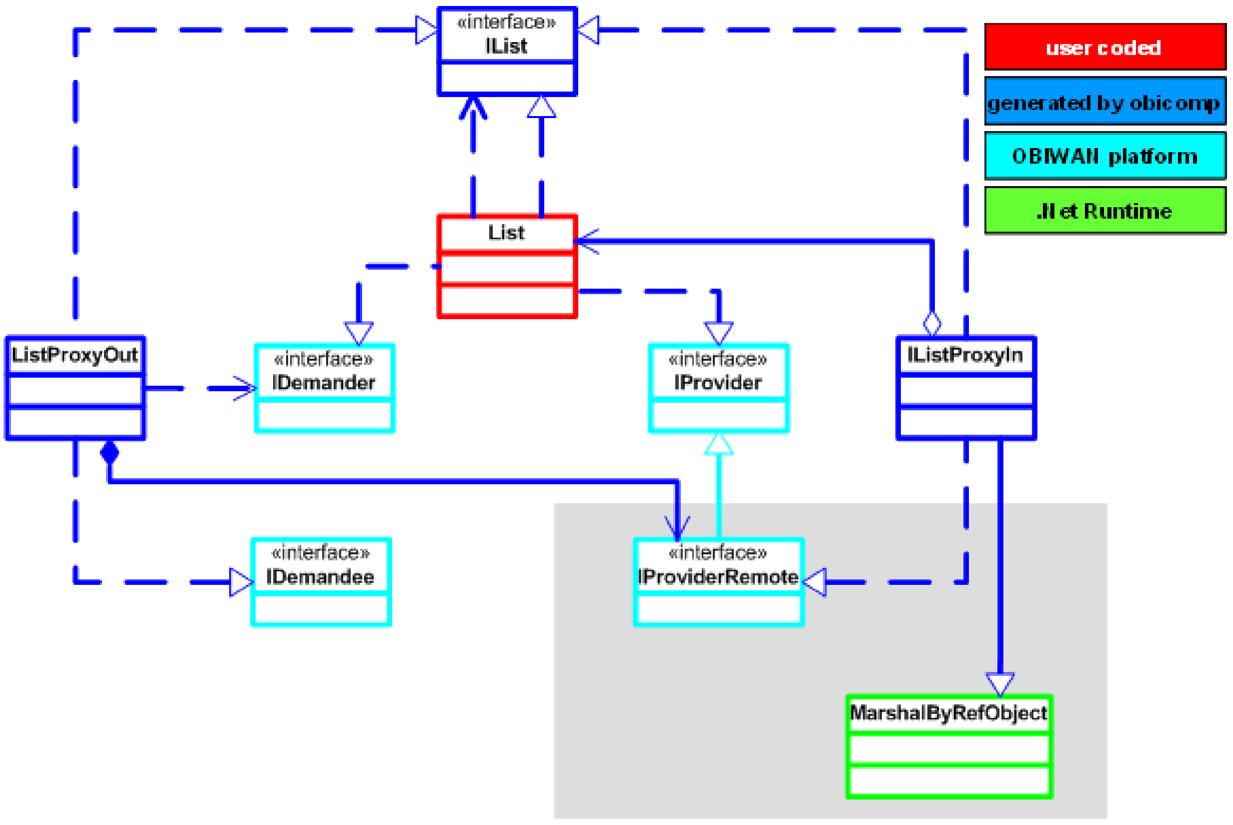


Figure II.3.4: UML class diagram for OBIWAN.Net implementation. The shaded part of the diagram highlights the main differences w.r.t. OBIWAN.Java.

reflection mechanisms w.r.t. inheritance, field types, public methods, and custom attributes. The `obicomp.net` compiler can analyze classes coded in any language supported by .Net. However, currently, it only generates C# code. Nonetheless, supporting other languages (e.g., VB.Net) is only required for class extension. OBIWAN library and proxy generators need not be changed since .Net CLR executes byte-codes, regardless of the source language used.

Classes and fields are tagged (by the programmer) with a special attribute (`[OBIWAN]`) that is accessible to reflection. This informs `obicomp.net` of classes to be extended and fields to be replicated incrementally.

Generation of proxy classes is performed by an autonomous class that creates the source files and writes code that implements the patterns described. After this, the proxy classes are compiled using `csc`.

Extension of an application class (e.g., `List`) is also performed by source code augmentation, i.e., by a class that inserts and modifies text in the class source file. All code generated by `obicomp.net` is enclosed in special *regions* (with `#region/#endregion` tags) to ease debug. After this the extended class can be recompiled and the extension process is finished.

II.3.2 OBIWAN for Mobile Constrained Devices

The available Java and .Net virtual machines for mobile constrained devices (Java 2 Micro Edition or KVM, and .Net Compact Framework) have a number of limitations, when compared with the desktop versions, namely w.r.t. remote invocation mechanisms and object serialization. These are due, mainly, to the reduced capabilities of the devices and to minimize memory footprint of the virtual machines themselves.

Java 2 Micro Edition defines two profiles for mobile constrained devices: i) CLDC (Taivalsaari 2000; Taivalsaari 2003),[3] and ii) CDC (Courtney 2001; Courtney 2005). [4] CLDC has no support for Java RMI but provides support for web-services invocation. CDC offers an optional RMI package but this profile targets high-end PDAs and set-top boxes. Thus, for common PDAs and mobile phones, there is no support for RMI.

The .Net Compact Framework (Wigley et al. 2003) does not provide neither remote method invocation (i.e., .Net Remoting) on objects, nor general-purpose object serialization.

These limitations exclude some of the mechanisms upon which the implementation of OBIWAN desktop prototypes was based. To address these limitations, we extended the OBI-

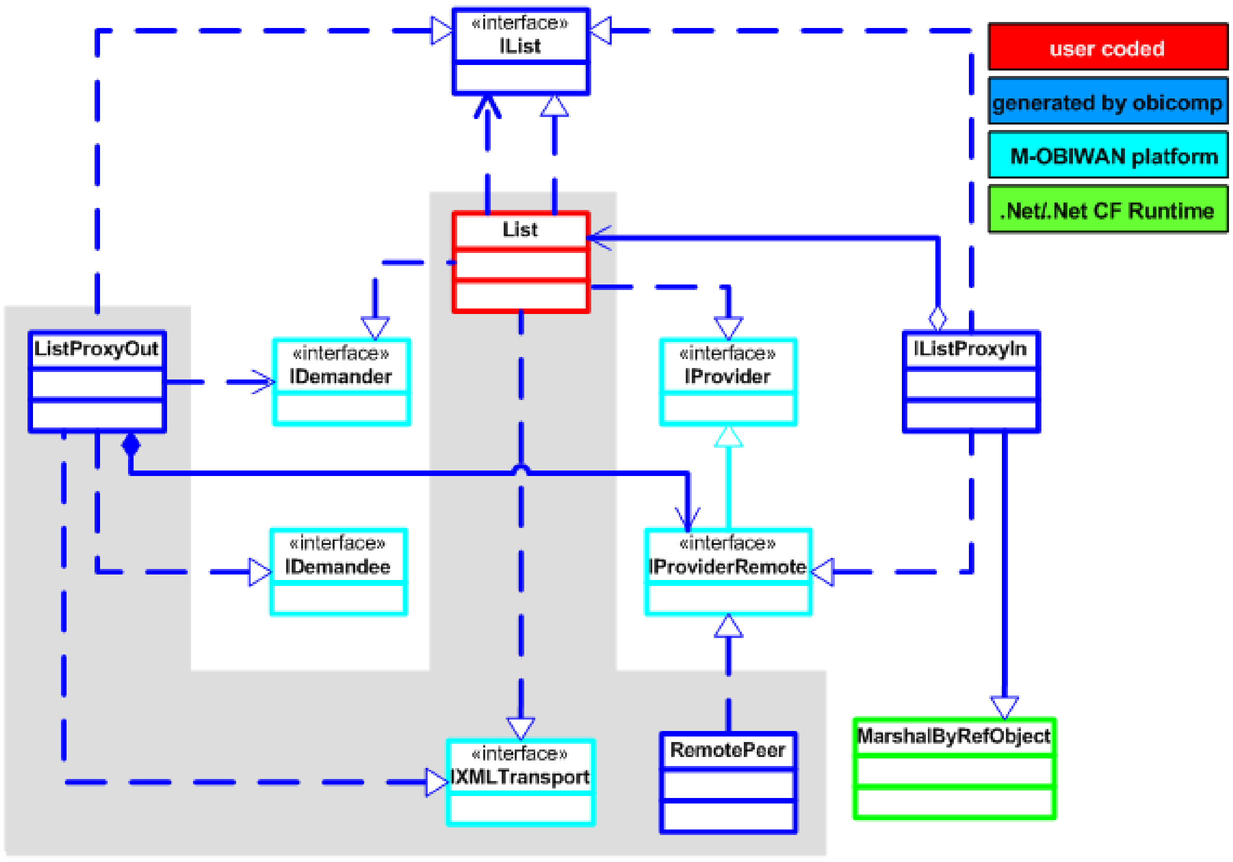


Figure II.3.5: UML class diagram for M-OBIWAN implementation. The shaded part of the diagram highlights the main differences w.r.t. desktop prototype OBIWAN.Net.

[3]Connected Limited Device Configuration.
[4]Connected Device Configuration.

WAN.Net prototype. Similar adaptations can also be applied on the CLDC profile. This would make OBIWAN portable to the complete universe of J2ME (as opposed to an implementation based on RMI).

II.3.2.1 M-OBIWAN

In order to run OBIWAN on mobile constrained devices, we developed a specific prototype M-OBIWAN, based on OBIWAN.Net. It runs on top of .Net Compact Framework (although nothing prevents it from running on .Net Framework, in a desktop environment, as well). Currently, M-OBIWAN only allows object replication, not direct invocation on remote objects from mobile constrained devices.

The diagram in Figure II.3.5 shows the result of class extension, by `obicomp.net`, using the same example class (`List`). The main differences from OBIWAN.Net are highlighted, and explained next: i) the use of a web-service bridge encapsulated by `RemotePeer` classes, and ii) XML serialization using object-*wrappers*, encapsulated by code implementing interface `IXMLTransport`.

II.3.2.1.1 Web-Bridge

Since the communication services of .Net CF provide basic support for web-service invocation, M-OBIWAN includes a web-bridge component to mediate access from mobile constrained devices to desktop computers. This is depicted in Figure II.3.6.

Desktop computers (when they act as servers) are unaware of the differences of clients, i.e., whether they are mobile constrained devices or not. They behave in the same manner as described earlier, i.e., they are invoked via .Net Remoting services. The web-bridge masks this heterogeneity by performing, w.r.t. the servers, as any other client would. In essence, the web-bridge serves as an intermediary between a proxy-out and its corresponding proxy-in. This is performed transparently w.r.t. both of them.

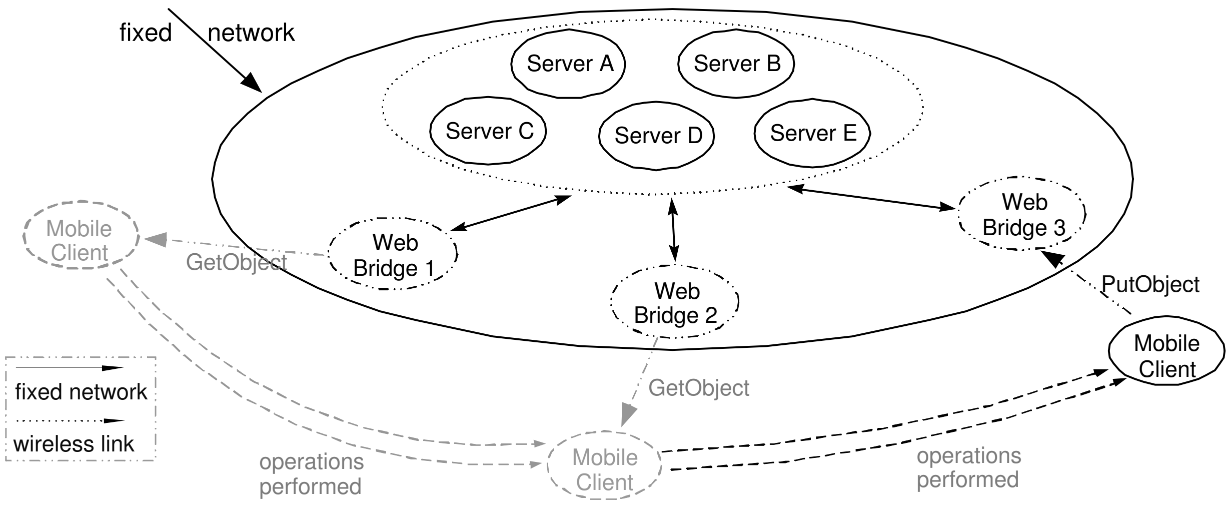


Figure II.3.6: M-OBIWAN Web-bridge mediation.

```
Wrapper
 GetObjectWS( string host, string name, int mode, int depth );

void PutObjectWS( string host, string name, Wrapper wrapper,
 int mode, int depth );
```

Table II.3.1: Web-Services provided by the M-OBIWAN web-bridge.

The web-bridge is a set of web-services (see Table II.3.1) running on top of Internet Information Server (available at desktop computers). Mobile constrained devices simply act as web-service clients. Web-bridges may interact and relay request to others.

The web-services mimic the methods of interface `IProviderRemote`. The differences are that exchanged objects are of a fixed type (`Wrapper`), and that explicit references to host and object name (or objectID) are required. This information is explicitly needed by the web-bridge in order to invoke the corresponding proxy-in. In desktop implementations, this information is implicit in remote references. However, as stated previously, these are not supported in .Net CF.

Thus, when a proxy-out wants to invoke a proxy-in, it must instead invoke a `RemotePeer` object that also implements interface `IProvider`, and that stands in the place of a remote reference of .Net Remoting. This `RemotePeer` object knows the identification of the proxy-in counterpart and provides that information (along with the other arguments it receives) when it invokes the corresponding web-service method.

The code inside the web-services must, in turn, obtain a remote reference to the corresponding proxy-in by invoking the remote method `GetProxyByName` on the intended server. After that, it can invoke the intended method (`get` or `put`) on the proxy-in obtained, with the arguments received. The web-bridge is also responsible for converting data between the different serialization formats involved (binary, used with RMI to communicate with desktop nodes; and wrapper-based, used with SOAP to communicate with mobile constrained devices).

The web-bridge in M-OBIWAN is stateless. Thus, the mobile constrained device may reconnect later using a different web-bridge. This allows mobile constrained devices to operate in environments with connectivity restricted to a local area.

Wrappers are used to serialize object replicas in a way that circumvents limitations of .Net CF, and are described with further detail in the next section. Each class is automatically extended with code that is able to replicate and update its state using wrappers. The web-bridge exchanges only wrappers with the mobile constrained devices.

II.3.2.1.2 XML Object Serialization

Object serialization in .Net CF is limited to the minimum required to support the invocation of web-services. In particular, there is no support for binary serialization. XML serialization is limited to public fields of objects.

It is not acceptable to force application programmers to expose all object fields as public (that would be required), in order to be able to replicate them to a mobile constrained device.

Furthermore, XML serialization handles references in a very simplified manner, as it does not handle full SOAP formatting. Referenced objects are simply included in referring objects, forming an XML hierarchy. Therefore, object identity is not enforced since the same object may be serialized more than once, and it does not handle cyclic subgraphs.

Thus, M-OBIWAN also includes specific code to support full object serialization in XML, preserving object identity, via the automatic implementation, for each class handled by OBIWAN, of a specific interface `IXMLTransport`, presented in Table II.3.2.

```
Wrapper toXML( void );
void fromXML( Wrapper wrapper );
```

Table II.3.2: Interface `IXMLTransport`.

Methods `toXML` and `fromXML` manipulate instances of class `Wrapper`. Wrapper objects carry class type information, and an array of `WrapperElement` objects, one for each field being serialized. These elements may contain primitive types or references to other `Wrapper` objects. This is a better option than simply creating, for each class, a *mirror* class with all its fields declared as public, as this would still not preserve object identity, and would hinder the programming model.

Wrappers are created and unwrapped, only by the web-bridge and the mobile constrained devices. Both application classes and proxy-out objects know how to handle wrappers. After a subgraph of objects has been replicated (returned by method `get` of a proxy-in), it is returned to the web-bridge. The web-bridge then invokes method `toXML` on the object that heads the subgraph to convert its wrappers that are serialized correctly to XML. This process is reversed when objects are being sent back to desktop nodes as updates.

Object Wrapping: With application classes, after the wrapper is created, it is filled with fields of primitive types (including the objectID). Fields of reference-types are wrapped by invoking method `toXML` recursively. No depth parameter is needed since it is assumed that all objects are to be converted to wrappers. The techniques employed to handle subgraphs converging to the same object, and cyclic subgraphs, are the same employed in object replication, as described in detail in Section II.3.3.8. References to objects already wrapped are replaced by their objectID.

Reference-fields of type `IProviderRemote` (both in object replicas and proxies-out, that target a proxy-in) are actually *remote references*; thus they are ignored, since they are not supported in .Net CF. In the case of reference-field `demander`, within a proxy-out object, it is converted to the objectID of the referenced object.

Object Unwrapping: Before a wrapper is unwrapped, its class type information is consulted and an instance of the appropriate class is created. Then, method `fromXML` is invoked on the instance, with the wrapper provided as argument. Before reference-fields are unwrapped, the appropriate objects are instantiated. Method `fromXML` is invoked recursively until all wrappers have been unwrapped and the corresponding objects instantiated. Whenever an object is instantiated, it is registered in the `OBIRep`. This is a step equivalent to customization of serialization in the desktop prototypes (described in Sections II.3.1.1 and II.3.1.2).

Before unwrapping a proxy-out, a check is made to verify if the corresponding object has already been unwrapped to get a reference to it. When a proxy-out is unwrapped it gets a reference to its demander, from the `OBIRep`, using its objectID that was included in the wrapper.

Reference-fields of type `IProviderRemote` (that were not wrapped) are set to reference a new object (of class `RemotePeer`) that includes the objectID of the object. Its implementation (methods `get` and `put`) invokes the web-bridge, in turn.

The remaining of the code generated by OBIWAN, that performs object-fault handling and incremental replication is unaware of the differences w.r.t. serialization and communication. They simply invoke methods of interface `IProviderRemote`.

II.3.2.1.3 API Transparency

The OBIWAN API is unchanged to application programmers in M-OBIWAN. However, in M-OBIWAN, communication is delegated to invoking web-services, instead of remote objects directly. Programmers never invoke web-services explicitly. A configuration file stores the web-bridge address.

Thus, in addition to OBIWAN.Net, `obicomp.net` only has to extend classes, by automatically generating code that implements interface `IXMLTransport`. The rest of the actions are performed by the web-bridge and the M-OBIWAN library.

II.3.3 Common Implementation Aspects

This section describes the details of relevant implementation aspects of the OBIWAN core that are common to all the prototypes.

II.3.3.1 Data Structures

For performance reasons, InProp and OutPropList structures are actually implemented in two levels as extended hash-tables (named `OBIRep`) of lists. In each process, there is an extended hash-table that maintains a structure (an `OBIRep` element) for each object that has been replicated *from* and/or *to* other processes. In this sense, the OBIRep functions as a top-level object table (w.r.t. OBIWAN) of the process.

Each `OBIRep` element stores: i) a direct reference to the object in that process, ii) one list of InProp entries, and iii) a list of OutProp entries (both w.r.t. the object). Thus, InPropLists and OutPropLists are implemented separately, each one concerning only a single object (see Figure II.3.7). This approach simplifies and speeds-up the most common task. It consists in accessing, regarding a particular object, a specific InProp or OutProp entry, in the context of an on-going replication operation among two processes.

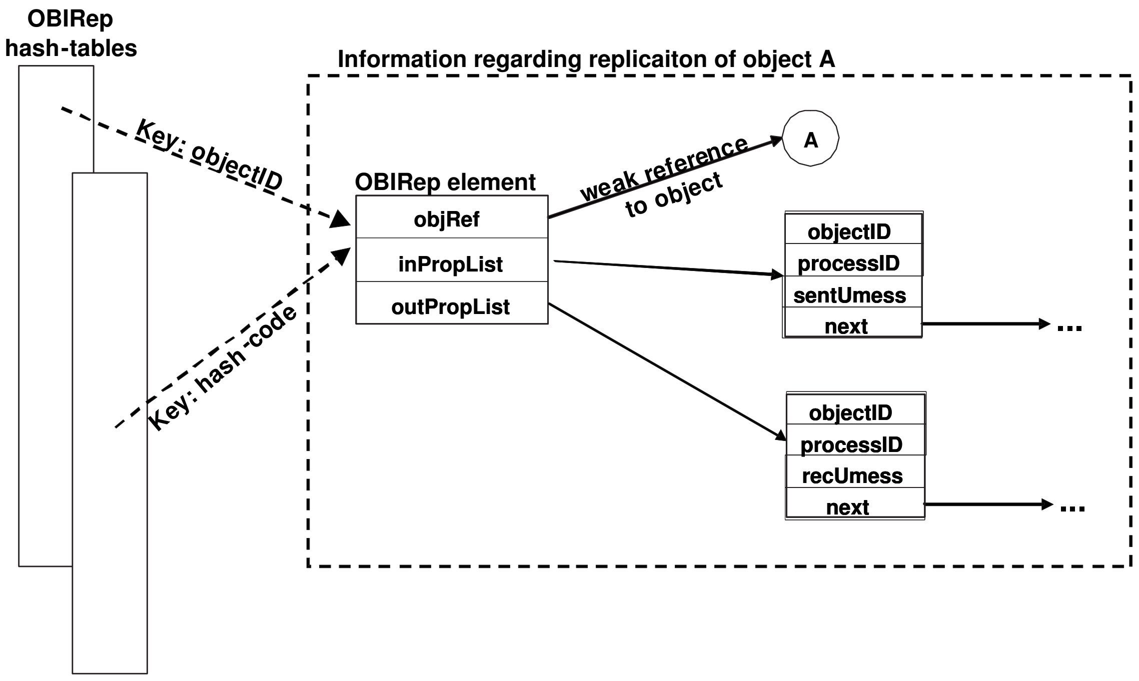


Figure II.3.7: Internal structure of `OBIRep` elements. Example depicted w.r.t. to an hypothetical object A.

The hash-tables are extended in the sense that they allow that references to objects be retrieved from it, using one of two alternative keys: hash-code and objectID. The implementation of `OBIRep` actually contains two hash-tables, each referencing the same `OBIRep` elements, with each of the mentioned keys.

When an object is first registered in the `OBIRep`, because it is being replicated (*in* or *out*) an entry (i.e., an `OBIRep` element) regarding its existence is created and inserted in the `OBIRep` hash-tables. This is performed using the objectID (created by OBIWAN) field as key in one of hash-tables and, in the other, using the hash-code (provided by the virtual machine) of the object replica. This allows access to information concerning the replication of an object, both using the objectID field (e.g., to check if an object has been replicated earlier) and directly via a reference to a replicated object.

InProp and OutProp lists in a `OBIRep` element are initially empty. Each entry stores, in the context of replication, and as presented in Chapter I.2: i) the objectID of the object, and ii) the process counterpart (where the object was replicated *from* or *to*). There is one entry in these lists for each process that the object has been replicated *from* and/or *to*. Other implications of these structures w.r.t. memory management are described in Part III.

II.3.3.2 Object Identity

ObjectIDs are attributed to objects when they are involved in a replication operation, to a different process, for the first time (recall Section II.2.6.3). This includes either replicating the object to another process, or creating a proxy-out/proxy-in pair, targeting the object, for ulterior replication. This gives the object a *name* that is shared by all replicas of the same object that may be created in the future.

This operation is performed lazily. There is no need to generate an objectID for an object that is not replicated in, nor referenced from, another process. The objectID of an object or proxy is exposed via an interface that is automatically implemented. This objectID is also used, when resolving a proxy-out, to check if the corresponding object was already replicated into the process.

Each process maintains a global counter (an integer) that is provided and incremented each time a new objectID is required. This value is unique within a process and, if object graphs are persistent, it is also made persistent to preserve uniqueness.

ObjectIDs must be globally unique (i.e., w.r.t. every other process), and must be generated independently by each process (i.e., without involving communication among processes). To this effect, in OBIWAN implementation, objectIDs combine a 32-bit ID number fixed to each process, with the current value of the object counter of the process (also limited to 32 bits), to compose a 64-bit globally unique objectID. Whenever possible, a process ID may be derived from the IP address of the computer it is running on.

II.3.3.3 Interface IProvider

Middleware code contained in methods `get` and `put` of interface `IProvider` has access to the private fields of the object's class, since classes are automatically extended in order to implement the interface. Therefore, this code can manipulate the private fields of the object where the methods are invoked, and to other objects of the same class.

Methods of `IProvider` rely on a replication context set-up before they are invoked. This replication context includes selecting the proper type (InProp or OutProp entries) in `OBIRep` data structures, and targeting the initial object to be replicated or updated. This takes into account the process counterpart involved in the replication operation, and the replication mode chosen. This set-up is performed by methods of the OBIWAN API, following the rules presented in Section II.2.6.9, whose implementation is presented in Section II.3.3.7.

II.3.3.3.1 Method `get`

Method `get` is invoked on objects when they are being replicated to another process (in `SERVER_MODE`)[5] or when objects are being sent as updates to another process (in `CLIENT_MODE`). This determines which propagation list in the `OBIRep` element will be used primarily. The one containing OutProp entries is used in `SERVER_MODE`, and the one with InProp entries in `CLIENT_MODE`.

When method `get` is invoked on an object it first checks if `TRANSITIVE_CLOSURE` replication mode was required. If it has not, then it must check the parameter for replication depth. If it

[5]Replication modes are implemented as `constants` in C# and `static final` fields in Java, e.g., `OBIWAN.ReplicationModes.CLIENT_MODE`.

is zero, then the object is not replicated and method `get` creates a proxy-out/proxy-in regarding the object as described in Chapter II.2.

If the object has already been replicated, or if a proxy-pair regarding it, has been created before to any other process, the object already has its objectID field initialized. If not, it is created then and a corresponding `OBIRep` element is created. The objectID is then passed to the proxy-out constructor. After the proxy-out/proxy-in pair is created, the method simply returns the reference to the proxy-out (that will stand as the object replica).

If the method continues, method `get` must determine if flag `REFRESH` is included in the replication mode. If this flag is not included, then, method `get` will not replicate objects that were already replicated to the same process previously. Otherwise, this test is never performed.

When this test is needed, it is performed by checking in the `OBIRep` (using its own hash-code as key), for an entry regarding that process, in the propagation list. If such entry already exists, then a proxy-out/proxy-in, regarding the object, is returned instead as described earlier.

If it has not, it inserts an entry in the propagation list, in the corresponding `OBIRep` element, and initiates the replication of the object: i) if required, creates a new instance of the object's class (via a default constructor) and copies its objectID, ii) copies all fields of primitive types and reference-types not handled by OBIWAN (that are thus regarded as private to the object being replicated), iii) invokes method get (recursively) for each field of reference-types handled by OBIWAN, passing the same parameters as received, only with decremented replication depth, and iv) if `CLUSTERING` is not activated, creates a proxy-in for the object and sets it as the replica's provider (using interface IDemander).

This delegates, by recursion, the replication of the referenced objects. For each of the replicated objects, if it is a proxy-out, it sets its demander (checking and invoking interface IDemandee) properly.

If `REFRESH` mode is used, OBIWAN code must force the replication of objects that were already replicated to the same process. Nonetheless, it must still ensure that the same object is not replicated more than once in the same replication operation. This accounts correctly for subgraphs converging to the same object. It also handles cyclic subgraphs correctly. This is achieved using an auxiliary `ObiRep` (named `TempProp`) that is also checked and updated when an object is being replicated. If it has already been so, in the context of the current replication operation, the already created replica is reused (instead of creating a new one), and replication along that path terminates.

If different paths converge to the same object, reaching it with different replication depths, only the depth of the first path reaching it is honored, i.e., the first branch reaching an object determines if it is to be replicated or not.

II.3.3.3.2 Method `put`

Method `put` is invoked on objects when they are being refreshed (in `CLIENT_MODE`), or when they are being updated with replicas modified in another process (in `SERVER_MODE`). This determines which list in the `OBIRep` element will be used primarily. The one containing OutProp entries is used in `SERVER_MODE`, and the one with InProp entries in `CLIENT_MODE`.

While method `get` creates replicas of the objects, method `put` must be applied on an object that already exists (otherwise, it could not be invoked). Thus, whenever an object field, previously holding a `null` reference, is updated with a non-null value, it is being modified to reference either a pre-existing object, or a newly replicated one. The last case raises the issue of object promotion that will be addressed later in this section.

When method `put` is invoked on an object, it first checks if `TRANSITIVE_CLOSURE` replication mode was required. If it has not, then it must check the parameter for replication depth. If it is already zero, then the object is not updated, and execution of method `put` simply terminates (method `put` has no return value). Replication mode `CLUSTERING` has no influence in the behavior of method put.

If execution continues, when method `put` is invoked on an object (e.g., A), a replica to that object is passed as argument (e.g., A'). Method `put` will update the fields of primitive types, and of reference-types not handled by OBIWAN, of object A simply by copying them from replica A'. W.r.t. to fields of reference-types handled by OBIWAN, method `put` behaves, for each of them, as follows.

First, method `put` must check if the field in updated replica A', actually references proxy-out objects. If replica A′ references a proxy-out, method `put` attempts to translate it by checking the `OBIRep` with the objectID stored in the proxy. If the object already exists, object A is updated to reference it. If not, the proxy-out is kept, and its demander field is updated (via method `setDemander` of interface `IDemandee`, so that it references A), and object A is also updated to reference the proxy. Method `put` terminates along this path.

If the field of replica A′ references another updated replica, method `put` may continue. It must check if the object referenced by A, and the replica referenced by A′, concern the same object. If not, object A must be switched to reference *the* object that matches (i.e., has the same objectID) the replica referenced by A'. A reference to it can be obtained from the `OBIRep`. This applies an update on a reference field in object A, so that it now references a different object.

Either way, updating the object graph may continue simply by following the recursion, i.e., invoking method `put` on the referenced object, with the referenced replica, and decremented depth, as arguments.

In the context of method put, if `UPDATE` mode is not used, it is implied that internal fields of replicated objects should not be updated, except in the case when they originally held `null` references. This way, it only integrates new objects in the object graph, without refreshing or updating objects already replicated.

If `UPDATE` mode is used, then the graph containing the updates may have subgraphs converging to a single object, or cyclic subgraphs. To account for this, the process used is analogous to that previously described for method `get` with mode `REFRESH`. This is achieved using an auxiliary `TempProp` that is checked when a replica is being used to update an object. If it has already been so, in the context of the current replication operation, then method `put` along this path terminates.

Replication modes `REFRESH` and `UPDATE` are maintained for symmetry and clarity: one for API method `GetObject`, and the other for API method `PutObject`.

Replica Promotion: An object replica is considered promoted, when it is the first replica of that object to be instantiated in the process. From then on, this replica becomes *the* actual object (w.r.t. this process) and other replicas of the same object, received later, will always be used just to update it.

Object promotion takes place automatically when the replicated objects are instantiated. If they are not registered in the `OBIRep`, they are registered then. Thus, whenever method `put` consults the `OBIRep` to obtain a reference to an object that is being updated (i.e., for which an updated replica has been received from another process), this check is always successful. If it returns the replica (i.e., the object didn't exist before in this process), it is promoted, in the sense that it is integrated in the graph maintained by the process, available to applications. To account for this, when method `put` is invoked on an object to update it with itself, method `put` simply continues the recursion and does not modify the object.

The replication of promoted objects integrated in a subgraph being updated, is registered in InProp entries (instead of the OutProp entries), i.e., they are regarded as initially belonging to the process where the object graph was replicated from. This is ruled by the use of `SERVER_MODE` in the replication operation.

II.3.3.4 Serialization and Instantiation

Once created by the execution of method `get`, replicated subgraphs must be transferred to other processes. This is performed automatically by the underlying virtual machine. It takes place when a method that was invoked remotely (e.g., method `get` of a proxy-in) returns the object replicas. The same happens when the object replicas are provided as parameters to a method invoked remotely (e.g., method `put` of a proxy-in). Additionally, object transfer may also be subject to an intermediate step of XML wrapping, as in M-OBIWAN.

Upon reception of the subgraph at the other process, the comprised objects are either automatically instantiated by the de-serialization mechanism or during unwrapping. Registration of objects in the `OBIRep` and creation of InProp or OutProp entries, when needed, is performed by code automatically generated by OBIWAN. It leverages the possibility provided by both desktop virtual machines to customize the serialization/de-serialization process, with added behavior, without having to re-implement basic object serialization. In the case of M-OBIWAN, this is performed by XML unwrapping code that is also automatically generated.

II.3.3.5 Proxy objects

Proxies-in are implemented as non-serializable objects, i.e., objects that never leave the process where they were created. When proxies-out and object replicas reference an `IProviderRemote` object, they are in fact referencing a non-serializable object. Thus, these references are the ones that set the boundary of the serialization process. This is how, in OBIWAN, subgraphs are effectively replicated, without serializing the whole object graph maintained by the process.

II.3.3.6 Inheritance

If a class inherits from another class that was also extended by OBIWAN, the interface `IProvider` is implemented by each of them. However, invocations from other objects always invoke the most specialized implementation, i.e., the implementation of the sub-class at the end of the inheritance chain. Thus, the implementation of methods of interface `IProvider` in derived classes must perform an additional step: to invoke the same method on the superclass (possibly recursively). This preserves encapsulation in the sense that each class knows how to replicate itself, and avoids making all class fields protected (even those explicitly defined as private by the programmer), as well as redundant code. This also provides modularity, since if the implementation of the superclass changes, there is no need to change the implementation (i.e., re-generate the code) of the derived class.

When method `get` is invoked in a sub-class at the end of an inheritance chain, after it creates the object replica and sets its private fields, it invokes method `get` on the superclass of the object with flag `RE_USE_OBJECT` included in the replication mode. Prior to this, it must register the just created object replica in an inheritance-specific, lazily created, auxiliary OBIRep (named PropReuse).

Method `get` of the superclass, when invoked with flag `RE_USE_OBJECT` must get a reference to the same object replica already created, from PropReuse. If the superclass is also a derived class, this method will also invoke method `get` on its superclass and so on. Obviously, method `get` invoked with flag `RE_USE_OBJECT` need not return a value, since the object replica was already created. Thus, it returns `null`.

In method `put`, the recursive behavior is similar with an exception. There is no need neither for flag `RE_USE_OBJECT` nor PropReuse, since when the object being updated is invoked, it obviously already exists.

Similarly, method `updateMember` of interface `IDemander`, that is invoked when a proxy is being replaced by a replicated object, must also delegate its execution to each superclass.

To support inheritance, proxies-out and proxies-in also must implement all the public methods defined by all the classes in the inheritance chain. This way, any invocation of the proxy-out (via any of the implemented interfaces) will trigger object-fault handling.

W.r.t. to private fields storing the objectID of the object, and its provider, they should be shared by, and accessible to any derived class. This prevents each class in the hierarchy from having these private fields duplicated, since they are only needed once for each object, regardless of the length of the class inheritance chain. Thus, these members are defined as `protected` and are only defined in classes that do not inherit from other classes extended by OBIWAN.

II.3.3.7 API methods

Methods `GetObject` and `PutObject` of the API provided by OBIWAN, are implemented as static methods of the OBIWAN library class. The flags of the various replication modes are also implemented as bit-field public constants of the OBIWAN class.

As already mentioned in Section II.2.6.9, the complete execution of each one of the API methods delimits a replication operation. Methods `GetObject` and `PutObject` perform, if required by the replication mode(s) chosen, the sequential execution of methods `get` in one process, and `put` in the other, according to Table II.2.6.

For example, in the case of an invocation of method `GetObject`, with replication mode `REFRESH`, this results in the invocation of method `get` on the other process (by remote invocation of a proxy-in), and its result provided as parameter for the invocation of method `put` on a local object. Conversely, in the case of replicas being sent as updates to another process, method `get` is invoked locally, and method `put` will be invoked remotely on a proxy-in.

In the process where the API functions are called (a client, for this purpose), they set-up the replication context by: i) registering the identification of the other process involved in the replication operation and, ii) if necessary by the replication mode, by instantiating an empty TempProp to be used by methods `get` or `put` (v. Section II.3.3.3).

The OBIWAN API includes a method for registering objects with a well known textual name. It also includes overloaded utility methods `GetProxyByName` that can be invoked remotely. These methods receive a well known name, and return a proxy-in for the object that was registered with it. If the argument is a 64-bit objectID, the `OBIRep` is searched instead.

II.3.3.8 Class Extension

In order to be handled by OBIWAN, an application class must be extended and the corresponding proxy-in and proxy-out classes created. Class extension is automatic and consists in the insertion and modification of class fields, and the automatic generation of code implementing a number of interface and specific methods. This is performed by the `obicomp` compilers.

Classes: For each class (e.g., `List` for a simply or doubly-linked list) being handled by OBIWAN, `obicomp` must:

- Create the interface `IList` from the public methods of `List` and generate the code defining the interface.
- Modify class definitions to implement the necessary interfaces: `IList`, `IProvider` and `IDemander`.
- If the class does not inherit from another class handled by OBIWAN, add a protected field `objectID`, and implement method `getObjectID` that exposes it.
- If the class does not inherit from another class handled by OBIWAN, add a protected field `provider`.

- Modify the declaration (i.e., type) of reference fields whose types are classes handled by OBIWAN, so that they reference the corresponding interfaces (e.g., interface IA, etc., instead of class A, etc.).
- Modify method parameters and return types accordingly, as well.
- Generate code to implement methods of interface `IProvider`.
- Generate code to implement methods of interface `IDemander`.
- Generate special method(s) for serialization customization.

We need to ensure transparency in order that application code never manipulates references to proxy-out objects. This way, object comparisons and reference comparisons can be performed using the native mechanisms and operators of the underlying virtual machine.

Therefore, since each proxy-out is private to its `demander` (i.e., each proxy-out is only referenced by one object), OBIWAN must ensure that an object never returns a reference to a proxy-out, yet to be replaced. This is accomplished as described in the next paragraphs.

One protected boolean field is added to the class (`repIncomplete`) if it does not already inherit from any other class handled by OBIWAN. This flag is set to `true` when the object replica is being created, if it references any proxy-out. This only happens when replication depth reaches zero, or some object has already been replicated and is not being refreshed.

In every method that returns a reference to a class handled by OBIWAN (one that may actually reference a proxy), `obicomp` inserts a prologue that checks field `repIncomplete`. If it is `true`, it simply invokes a private method that triggers replacement of all the reference fields that are proxies-out. This is performed by invoking method `demande` on each proxy.

This action is performed only once for each object, and only for those objects at the boundary of a subgraph being replicated. This way, applications never manipulate references to proxies-out. They are unaware of their existence, only the OBIWAN middleware sees them.

Proxy-Out: To create a proxy-out class (e.g., `IListProxyOut`, the proxy-out corresponding to class `List`), `obicomp` must:

- Generate class definition implementing the required interfaces:
  - The one of the corresponding class (`IList`).
  - The ones from superclasses also handled by OBIWAN (a proxy-out must be able to stand-in for any superclass).
  - `IDemandee` from OBIWAN.
- Insert private fields `provider`, `demander`, and `replica`.
- Generate methods of interface `IDemandee`.
- Generate the methods of the class interfaces that start by invoking method `demande`, delegate on the replica when available, and invoke method `updateMember`.

Proxy-In: To create a proxy-in class (e.g., `IListProxyIn`, the proxy-in corresponding to class `List`), `obicomp` must:

- Generate class definition, indicating it should not be serialized, and enable it for remote invocation.
- Declare the implementation of the required interfaces:
  - The one of the corresponding class (`IList`).
  - The ones from superclasses also handled by OBIWAN (a proxy-in must be able to stand-in for any superclass).
  - `IProviderRemote` from OBIWAN.
- Insert private field `provider`.
- Generate methods of `get` and `put` of interface `IPRoviderRemote`.
- Generate the methods of the class interfaces that delegate on the object referenced by the proxy-in (`provider`).

In summary, the programmer only has to write the code of the application class. The class extension process and creation of proxy classes is automatic.

II.3.3.9 Support for Execution Migration

OBIWAN.Java and OBIWAN.Net provide support for the migration of execution flow through a specific interface named `IRestartable`; it is automatically implemented by `obicomp.java` and `obicomp.net`. Programmers just need to implement method `run`. In OBIWAN.Java, it extends the `java.lang.Runnable` interface, while in OBIWAN.Net, it serves as a `System.Threading.ThreadStart` delegate.

Since thread stacks are not first class objects (neither in .Net nor Java) the programmer must provide hints, regarding places in the code, in which agent's execution can be frozen, its state serialized, and transferred for ulterior reactivation upon arrival to another process.

Thus, at certain points of execution, the programmer must invoke method `checkpoint` of interface `IRestartable` (recall that all methods of this interface are automatically implemented).[6] The checkpoint method implements synchronization, so that it is safe to freeze the execution flow on an object, serialize its data, transmit it, and re-activate it in another process through the creation of a new dedicated object thread.

Prior to invoking the checkpoint method, it is the programmer's responsibility to set the object in a stable state that does not rely on stack frame information, i.e. the object can be restarted correctly (from an application's semantic point of view) in another process.

[6] A similar approach, using byte-code enhancement, is employed in other work (Handorean et al. 2005), which is also presented in Part IV w.r.t. adaptability.

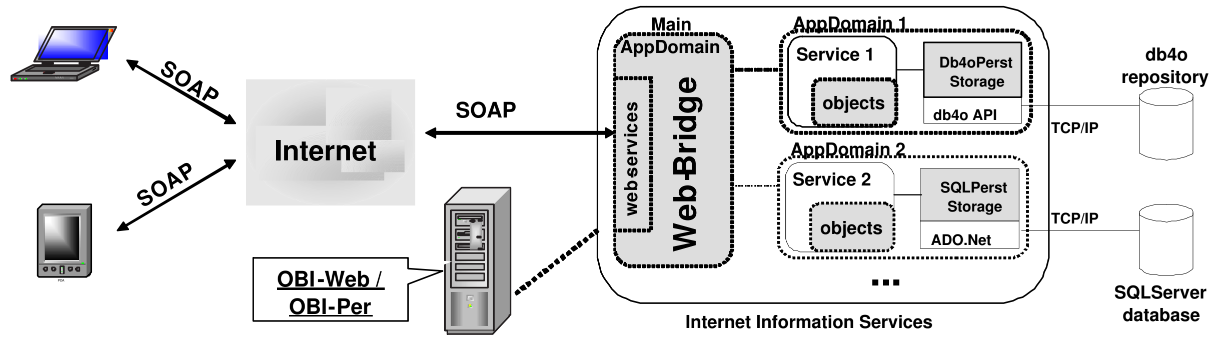


Figure II.3.8: OBIWAN Support for Application Servers and Persistence.

II.3.4 Support for Application Servers and Persistence

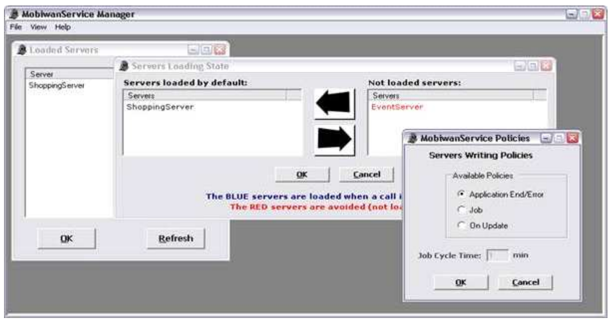
Figure II.3.8 presents, in greater detail, the implementation of OBI-web and OBI-Per. Applications running on client devices (PDA, laptops, etc.) replicate objects from an application server running the OBI-Web and OBI-Per prototypes. The relevant details of OBI-Web and OBI-Per are described in the next sections.

II.3.4.1 OBI-Web

We leverage the use of IIS to host and manage services that provide object repositories. We call them object services since they are not stand-alone processes. However, each of them exposes the remote methods of OBIWAN API, via its own object. The syntax of host names is extended so that an object service is identified as `<host>/<serviceName>`.

Thus, in the case of OBI-Web, a web-bridge that provides a number of object services is no longer stateless, as in M-OBIWAN. Nonetheless, the interactions between the web-bridge and object services, are performed as in the case of stand-alone processes (i.e., there are no direct invocations, from the web-bridge, of code inside the object services). The web-bridge simply maintains a correspondence between object service names and their application domains, in order to relay requests. It does not maintain information regarding individual objects. In particular, OBIRep structures are private to each object service.

Each object service runs within its own application domain (e.g., AppDomain1, AppDomain2 in Figure II.3.8), which provides isolation w.r.t. other object services in case of failure. Since the web-bridge and object services run on different application domains, communication between the web-bridge and each of them, must resort to .Net Remoting. Figure II.3.9 illustrates: i) the OBI-Web Service Manager, in charge of loading, unloading and configuring object services, and ii) the selection of a Visual Studio template to facilitate the development of OBI-Web object services.



Server: ShoppingServer

Servers loaded by default: ShoppingServer

Not loaded servers: EventServer

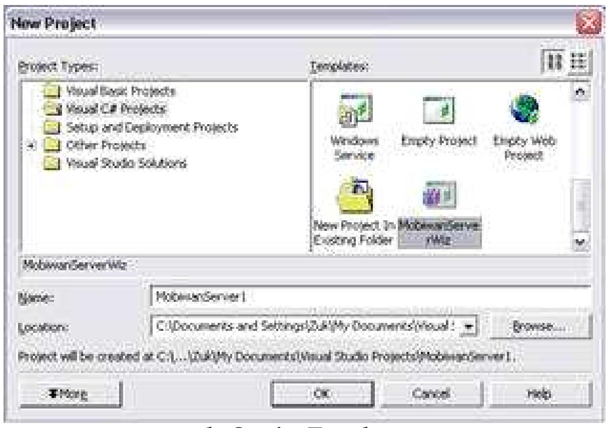
The BLUE servers are loaded when a call i

The RED servers are avoided (not lo

Available Policies: [✓ Application End/Error] Job On Update

Job Cycle Time: min

a. Service Manager.



MobiwanServerWiz

Name: MobiwanServer1

Location: C:\Documents and Settings\Zuk\My Documents\Visual

Project will be created at C:\...\Zuk\My Documents\Visual Studio Projects\MobiwanServer1.

b. Service Template.

Figure II.3.9: OBI-Web Service Manager and Service Template for Visual Studio.

II.3.4.2 OBI-Per

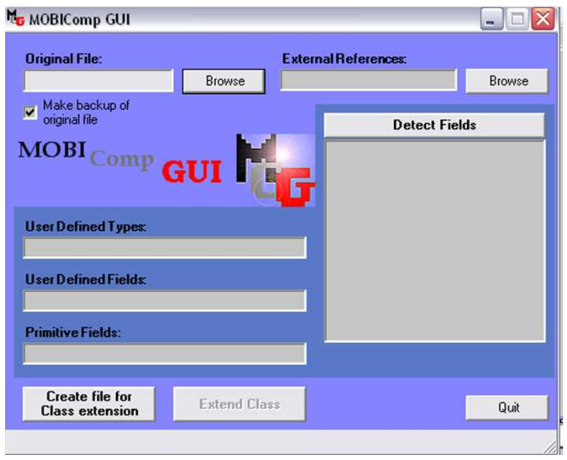
Object services in OBI-Web, and OBIWAN processes, may create the objects they manipulate and allow other processes to replicate, in a programmatic manner. Since this is time-consuming for programmers, OBI-Web automatically loads and saves object graphs, hosted by object services, using file-based persistence (by means of binary or XML serialization).

In alternative, object services may store object graphs persistently on object-oriented or relational databases. This support is provided by OBI-Per that extends the OBI-Web Service Manager and manages interaction with external persistence support. It defines an implementation of interface `IProvider` that communicates with an OBI-Per persistence manager to load and store objects. OBI-Per persistence managers are depicted in Figure II.3.8: `Db4oPerstStorage`, and `SQLPerstStorage`. They extend a common abstract class (`PerstStorage`), while adapting w.r.t. the specific details of a particular external persistence tool.

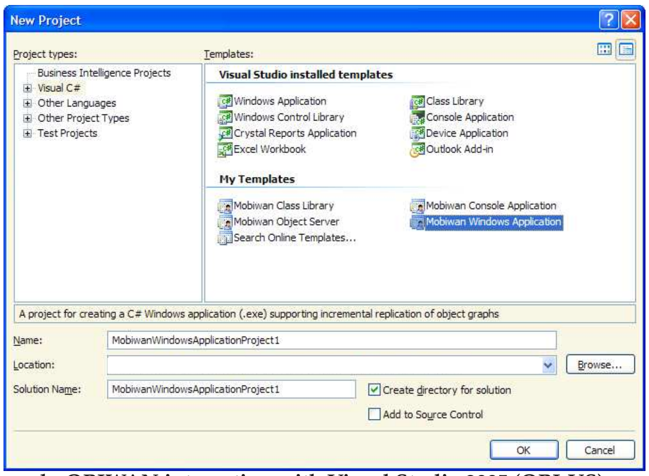
The OBI-Per implementation of interface `IProvider` returns returns customized proxies that resolve object-faults by loading the objects from persistent store. ObjectIDs are stored in persistent store and used as keys. When objects are loaded from persistent repository, they already have an objectID, which must have been given earlier. Therefore, to ensure uniqueness, objects are registered in the `OBIRep` of the object repository, although initially, without any associated InProp/OutProp entry.

II.3.5 Transactional Support

OBIWAN supports the incremental replication of large object graphs, including into mobile constrained devices. Additionally, it allows the creation of dynamic clusters of data, while providing hooks (through events) for the application programmer, to implement a set of application-specific properties. An example usage of such hooks is the transactional support described in (Santos et al. 2004). It is built upon the basic infra-structure and provides relaxed transactional semantics and updates dissemination.



a. OBIWAN Interactive Compiler (obicompGUI).



A project for creating a C# Windows application (.exe) supporting incremental replication of object graphs

Name: MobiwanWindowsApplicationProject1

Location:

Solution Name: MobiwanWindowsApplicationProject1

[✓] Create directory for solution

[ ] Add to Source Control

b. OBIWAN integration with Visual Studio 2005 (OBI-VS).

Figure II.3.10: OBIWAN Support for Application Development.

II.3.6 Support for Integrated Development Environments

Support for application development in OBIWAN was initially restricted to using `obicomp` tools. Figure II.3.10-a depicts `obicompGUI`, an interactive front-end for the OBIWAN compiler, that allows the developer to load classes, select which fields should be incrementally replicated, and generate code for class extension and proxies.

OBIWAN has also been incorporated in a commercial integrated development environment: Microsoft Visual Studio 2005. The OBI-VS plug-in targets OBIWAN prototypes executing on top of .Net (OBIWAN.Net, M-OBIWAN, OBI-Web, OBI-Per). It is depicted in Figure II.3.10-b with the project templates it provides for console and windowed applications, class libraries, and object services. In OBI-VS, all phases previously performed by `obicomp` tools (class analysis using reflection, parsing, source code generation and extension) have been adapted, in order to make use of the `System.CodeDom` name-space, and be triggered by `MSBuild` files.

The `System.CodeDom` name-space provides a set of classes that makes possible to create and manipulate a syntactic tree[7] representing the overall structure of, and syntactic elements included in, source files coded in .Net languages. `MSBuild` is the base tool for application compilation in the .Net Framework. It is used by both the command-line compilers (e.g., `csc`) and by Visual Studio 2005. It uses XML files to describe the contents of projects (`.csproj`) such as files, libraries and resources, as well as to describe the compilation tasks (`.targets`) necessary to build them.

`MSBuild` files are used to deploy OBI-VS in order to include class analysis and extension tasks, required by OBIWAN, in the application build process of Visual Studio. By default, project files reference a built-in compilation file: `Microsoft.CSharp.targets`. In the case of OBIWAN applications, project files reference a different compilation file that imports from `Microsoft.CSharp.targets`, and extends its behavior. After the project is built using the default compilation task performed by Visual Studio (`CoreCompile`), it loads

[7]Actually implemented as a directed acyclic graph for optimization purposes.

`OBIWANCodeGeneration`, a user-defined compilation task coded in C#, whose `Execute` method invokes OBIWAN code.

OBI-VS is activated during the build process when the meta-data attribute `[OBIWAN]` is encountered. A CodeDom tree is created for each class tagged with the `[OBIWAN]` attribute, using an open-source C# parser, based on the one included in Mono (Novell 2004).[8] It has been extended to handle code inside methods, instead of just signatures of methods and declarations of fields and properties. Class extension is performed by inserting the appropriate nodes (namely: declarations of `IProvider` and `IDemander` interface inheritance, fields, and automatically implemented methods with respective code) in the CodeDom tree, by following the approach described previously in Section II.3.3.8. Proxy generation is performed by creating a new CodeDom tree and filling it with the corresponding declarations and code.

Finally, the approach used to incorporate the OBI-VS plug-in for Visual Studio could also be used to integrate the OBIWAN.Java prototype with build tools such as **ant** (that uses XML files to express dependencies and build rules), and to specify and deploy plug-ins in Java-based integrated development environments, such as **Eclipse**.

Summary of Chapter: In this chapter, we described the implementation of OBIWAN in a number of prototypes (OBIWAN.Java, OBIWAN.Net, M-OBIWAN) targeting the Java and .Net virtual machines in different environments. The OBIWAN.Java and OBIWAN.Net prototypes, implemented in Java and C# respectively, execute on desktop and laptop computers in the context of wide-area distributed networks. The M-OBIWAN prototype is targeted at the .Net Compact Framework and executes on mobile resource-constrained devices (such as PDAs) used in mobile and pervasive computing. We described how the prototypes inter-operate and communicate over the network.

We presented in detail the implementation of the OBIWAN middleware core that follows a common pattern in all the prototypes: i) data structures managing replication, ii) OBIWAN interfaces `IProvider`, `IDemandee`, and `IDemander`, and iii) other relevant issues such as object identity, serialization and instantiation, and class inheritance. The code implementing the interfaces is automatically generated by `obicomp` compilers. Special detail was given to the extension of application classes, describing how interfaces `IProvider` (methods `get` and `put`) and `IDemander` are implemented. We also demonstrate how the code for proxies-out and proxies-in is generated.

We show how the limitations of existing virtual machines for mobile constrained devices (e.g., absence of remote method invocation and proper object serialization) are circumvented by using a communication bridge based on web-services, and automatic conversion of objects into wrappers, using XML. We also addressed the integration of OBIWAN with application servers (OBI-Web) and persistence tools (OBI-Per). Finally, we described how OBIWAN can be deployed within a commercial integrated development environment (e.g., Visual Studio), in order to fully automate application development in OBIWAN.

[8]An open source Unix version of the .Net runtime and compilers.

II.4 Evaluation

This chapter describes the evaluation of incremental object replication with OBIWAN, both quantitatively and qualitatively. The main part of the chapter is dedicated to performance measurements of incremental object replication of the OBIWAN prototypes. Then, we briefly describe some example applications developed on top of OBIWAN.

II.4.1 Performance Evaluation

The performance results are presented following the same structure adopted in the previous chapter. First, we present the performance results of the prototypes that target exclusively desktop computers (OBIWAN.Java and OBIWAN.Net). Then, we present the results of M-OBIWAN, OBI-Web, and OBI-Per together.

II.4.1.1 OBIWAN Desktop prototypes

In this section we present experimental results of incremental replication in OBIWAN.Java and OBIWAN.Net, with and without clustering with varying parameters. All the results were obtained with Pentium III and Pentium II PCs, with 128 MB of main memory each, connected by a 100 Mb/sec local area network, running JDK 1.3, and .Net Framework 1.0.

II.4.1.1.1 OBIWAN.Java

To study the performance of incremental replication in OBIWAN.Java, we performed several experiences of incrementally replicating into process P1 (that acts as a client), a list of objects from process P2 (that acts a server).

We use objects of several sizes, intended to be representative of small, medium, and large objects in common applications: 64 bytes, 1024 bytes and 16 Kbytes, respectively.

We use a list with 1000 objects that is created initially in process P2. Each element of the list, besides containing a reference to the next element, also holds a *byte* array as private data, that simply adds the corresponding payload to the object. In each experience, all the objects in the list are of the same size.

The application running in P1 starts by obtaining a reference to the head of the list (using a well-known name). Then, it iterates over the list, as it invokes a method on each object of the list, until it reaches its end.

When the object being invoked is not yet replicated into P1, OBIWAN automatically replicates it, and optionally a number of following objects using a certain depth. When an object is being replicated, the object payload (a byte array) is not explicitly copied (i.e., cloned), it is just referenced by the new replica. Cloning is unnecessary because the payload will be copied when the object replica is serialized.

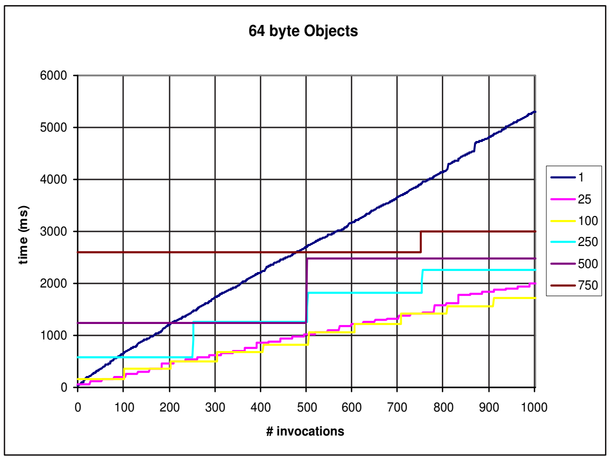
The depths used in the experiments are 1, 25, 100, 250, 500, and 750. The list is thus replicated into process P1, in several steps, each step replicating 1, 25, 100, 250, 500, or 750 objects at once.

The results are presented on the left-hand side of Figure II.4.1. The time values include the creation of object replicas, and proxy-out standing for the object that will be replicated in the next phase, as well as corresponding proxies-in. They also include the cost to serialize, transfer, and de-serialize all the object replicas and proxy-out, all with internal remote references to their corresponding proxy-in (its provider). In this case, each object can still be updated individually, and invoked remotely, since there is the creation of a proxy-in for each object being replicated.

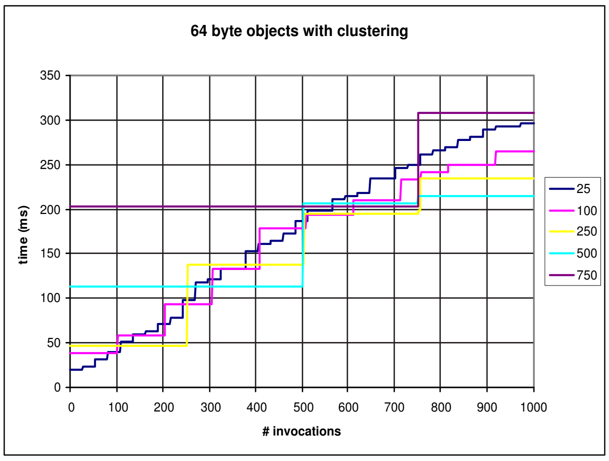
From the left-hand side of Figure II.4.1, we can observe that:

- The steps observed are due to the creation of replicas, proxies-in and proxy-out, and transference of replicas, and proxy-out. Each time an object-fault is detected, a number of objects are replicated before execution may resume. This delay appears in the Figure as a step.
- The creation and transference of replicas along with the corresponding proxy-in/proxy-out pairs is more significant than object invocations. Naturally, the actual cost of iterating over the list's elements, once they are replicated, is negligible. When more than one object is replicated at once, there is no noticeable slope in the graph, during object invocations, just until the next time replication takes place.
- Although the incremental replication of one object each time is the most flexible alternative, it is the least efficient for large number of invocations, especially for small and medium sized objects. This is due to network latency, that imposes a penalty for every access, regardless of the amount of data being transferred.
- The incremental replication of 25 to 100 objects at once is the most efficient alternative, w.r.t. to all object sizes. Depths in this range balance serialization complexity and network latency (more details in Section II.4.1.1.1.1). With these replication depths, the difference between replicating objects of 64 or 1024 bytes becomes almost insignificant.
- The incremental replication of 750 or 500 objects at once is less efficient because of the high cost of creation and transference of the corresponding replicas and proxy-out/proxy-in pairs. It imposes a more noticeable delay to applications, both initially and every time objects are replicated. Furthermore, the probability of some of the replicated objects never actually being invoked is higher. This wastes work, and memory at the client, specially in the case of large objects.

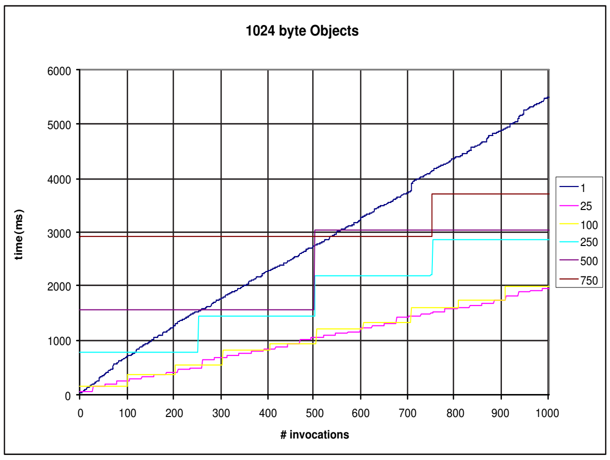
In conclusion, to improve the execution time of applications, when only a part of the objects will be effectively needed, it is clearly advantageous to replicate the objects incrementally as



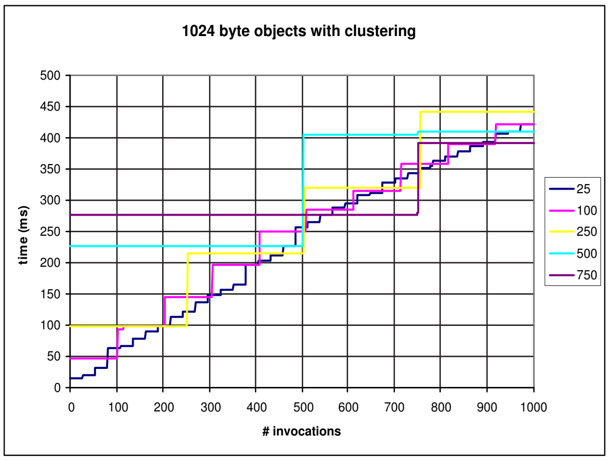
a. OBIWAN.Java Performance 64-byte.



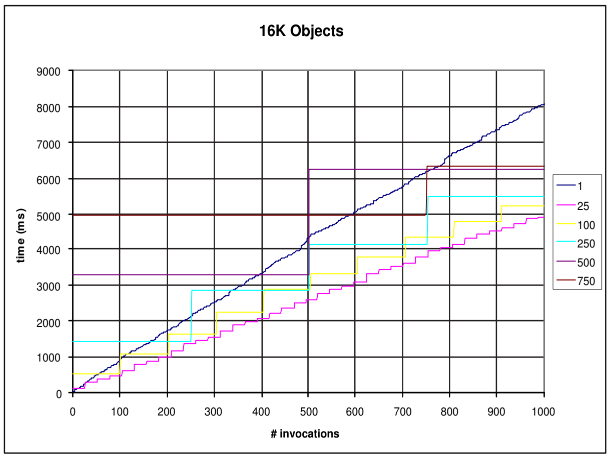
b. OBIWAN.Java Performance 64-byte clustered.



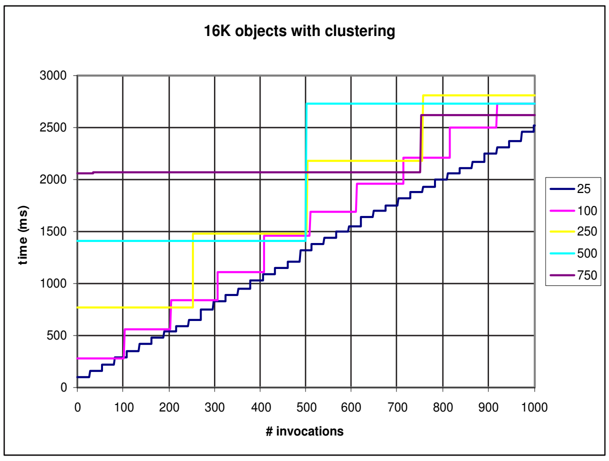
c.OBIWAN.Java Performance 1024-byte.



d. OBIWAN.Java Performance 1024-byte clustered.



e.OBIWAN.Java Performance 16 K-byte.



f. OBIWAN.Java Performance 16K-byte clustered.

Figure II.4.1: OBIWAN.Java Performance Results.

they are invoked. However, there should be replicated more than one object each time, i.e., a small number of them (e.g., 25), sufficient to amortize latency effectively, without incurring the application in significant delay.

II.4.1.1.1.1 Performance of Cluster Replication

We also study the performance of incremental replication with object clustering. The experiments are similar. The list and object sizes are the same as in the previous experiments. The application running in process P1 behaves identically, i.e., it simply iterates over the list, invoking a method on each object, until it reaches the end if the list.

Thus, when the object being invoked is not yet replicated, the system automatically replicates the next 25, 100, 250, 500, or 750 objects. The difference is that objects are replicated in groups, i.e. clusters with several sizes: 25, 100, 250, 500, or 750 objects.

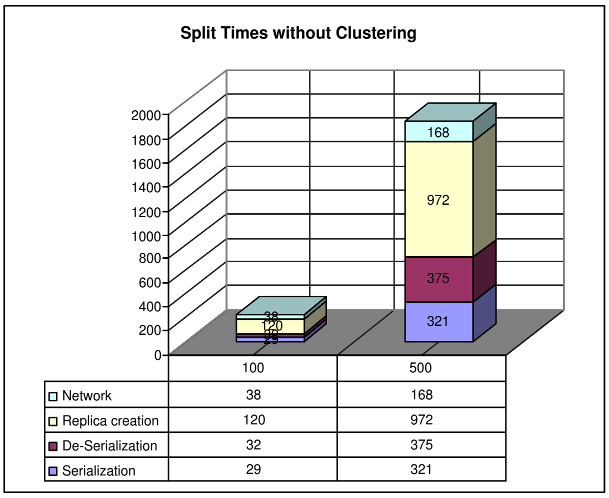
This means that, for each of these clusters, all objects are replicated as a whole. There is only one proxy-in/proxy-out pair being created. Consequently, each object can not be individually updated but object replication is faster. This is so since much fewer objects are created and transferred, and the number of references among them is also smaller.

The results are presented in the right-hand side of Figure II.4.1. In each case, the time values include the creation and transference of all the replicas; notice that, for each cluster, only a single corresponding proxy-out/proxy-in pair is created.

From Figure II.4.1, we can conclude that:

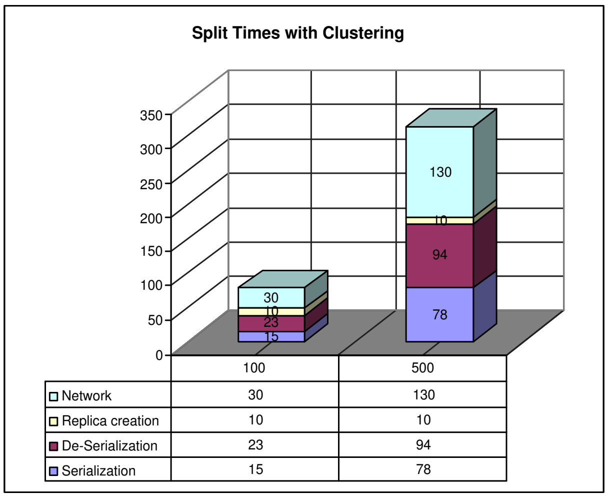
- The performance results are much better, when compared to the previous section.
- This improvement is due to the fact there is only one proxy-out/proxy-in pair being created and transferred for each cluster. Thus, there is significant less object creation and transference (about half), and reference-setting between objects. The most significant performance cost is data serialization (done by the Java virtual machine) and network communication.
- The improvement also suggests that the creation and initialization of proxies-in and the serialization and de-serialization of remote references (that is practically absent in clustering mode) is responsible for a significantly fraction of replication time.
- When compared to the previous section, the performance results are not that sensitive to the amount of objects being replicated each time (i.e. the curves are closer). This is due to the same reason as in the previous item.

Discussion In order to better understand the reasons for the differences observed in Figure II.4.1, w.r.t. object replication with and without clustering, we instrumented OBIWAN code and customized Java RMI, in order to obtain precise timings for each relevant aspect of object replication (replica creation, serialization, de-serialization and network transfer), as well as bandwidth usage.



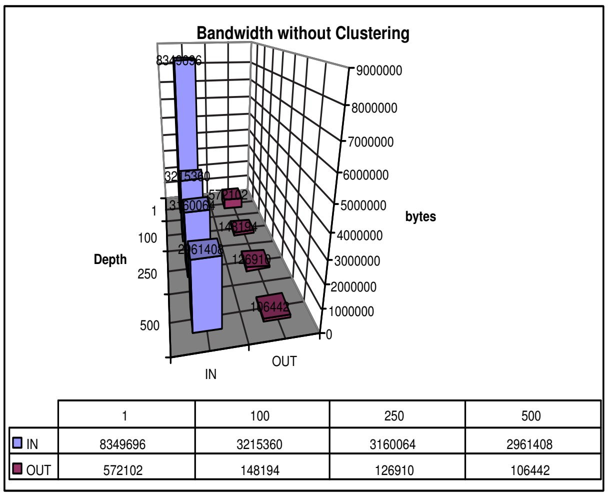
	100	500
▫ Network	38	168
▫ Replica creation	120	972
■ De-Serialization	32	375
■ Serialization	29	321

a. Split Times without clustering.



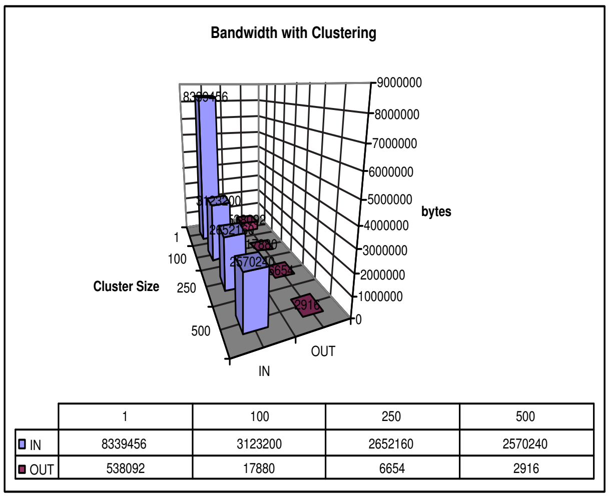
	100	500
▫ Network	30	130
▫ Replica creation	10	10
■ De-Serialization	23	94
■ Serialization	15	78

b. Split Times with clustering.



	1	100	250	500
■ IN	8349696	3215360	3160064	2961408
■ OUT	572102	148194	126910	106442

c. Actual Bandwidth usage without clustering.



	1	100	250	500
■ IN	8339456	3123200	2652160	2570240
■ OUT	538092	17880	6654	2916

d. Actual Bandwidth usage with clustering.

Figure II.4.2: OBIWAN.Java performance (Result Decomposition for 1024-byte objects). Bandwidth is measured from the point of view of client process (P1): i) IN - from P2 to P1, ii) OUT - from P1 to P2.

RMI customization was performed by installing an alternative `RMISocketFactory` that uses an extended socket (`CountSocket`) class. These sockets use extended `FilterOutputStream` and `FilterInputStreams`. They sit on top of existing input and output streams (of the underlying virtual machine) and simply count the number of bytes exchanged via them.

Instrumentation of OBIWAN code to obtain timings of the different phases of object replication was automatically generated and consists of inserting code in the beginning and end of methods of interfaces `IProvider`, and `IDemandee` in objects, proxies-out and proxies-in. This code simply registers a time, with a short text description (e.g., differentiating proxies from objects). This additional code obviously introduces some overhead (w.r.t. to the findings on Figure II.4.1) but it is marginal, and it manifests rather evenly over the tested cases.

The results are portrayed in Figure II.4.2. The top graphs show time decomposition for the different phases of object replication for a single replication operation (either 100 or 500 objects at a time, with and without clustering). Object size is 1024-byte (the intermediate size) in all cases.

It is clear that, without clustering, replica creation, serialization and de-serialization, take more time and that this effect is cumulative and not linear (i.e., total times for replicating 500 objects are more than five-fold those of replicating 100 objects).

With clustering, replica creation is quite inexpensive and even difficult to measure. Serialization and de-serialization costs together surpass that of network transfer. All the three increase quasi-linearly from 100 to 500 object clusters.

Without clustering, network times are not much greater, since the object payload partially masks the cost of transferring the additional remote references. However, replica creation, serialization and de-serialization times are much greater (specially in the case when 500 objects are replicated). This is due to the large overhead introduced by creating and initializing proxies-in (that are Java `Remote` objects), serializing an object graph containing so many remote references (100 and 500, respectively), and reconstructing them on de-serialization time.

The graphs at the bottom of Figure II.4.2 illustrate bandwidth usage in *bytes* when iterating the complete 1000 object list, replicating 1, 100, 250 and 500 objects at a time, with and without clustering. These measurements include not only traffic regarding actual object data being transferred, but also every other RMI-related traffic (remote invocations, remote reference set-up, etc.).

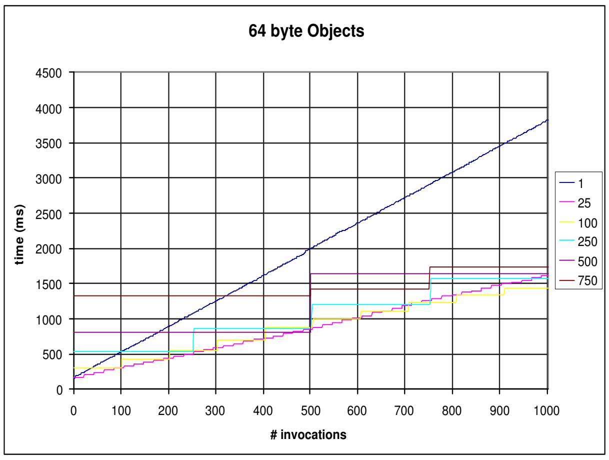
It is clear that incremental replication of one object at a time, clearly wastes bandwidth (uses between 2.8 and 3.2 times more than with 500 objects). In general, replicating more objects at a time reduces the total inbound (from P2 to P1) bandwidth used. This is because there are fewer remote invocations taking place (2, 4, 10, or 1000). The lower bound value would be just slightly higher than 1024000 *bytes*, i.e., the combined total payload of all objects in the list. RMI is less efficient than this optimum, mainly due to the insertion of type and remote invocation information in serialized data.

The graphs also show that the outbound (from P1 to P2) bandwidth usage if far greater without clustering, than with clustering. The significant values are those of replicating 100, 250, and 500 objects each time. It uses between 8.2 and 36.5 times more bandwidth. This is due to RMI traffic "behind-the-scenes" regarding remote references being de-serialized and set-up.

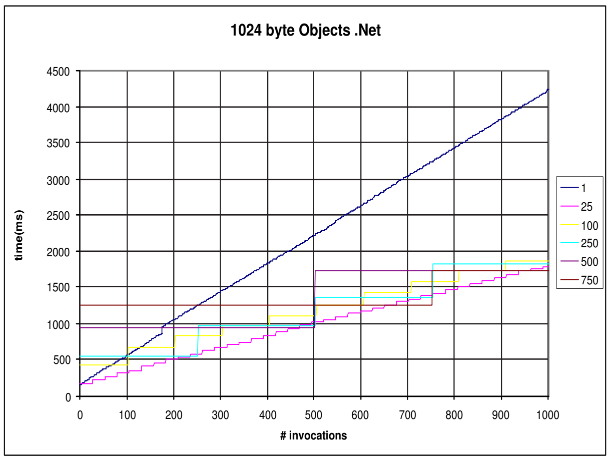
II.4.1.1.2 OBIWAN.Net

We now present the performance results of the OBIWAN.Net prototype. They are briefer than those of OBIWAN.Java, since they exhibit mainly the same tendencies observed for OBIWAN.Java, and discussed earlier.

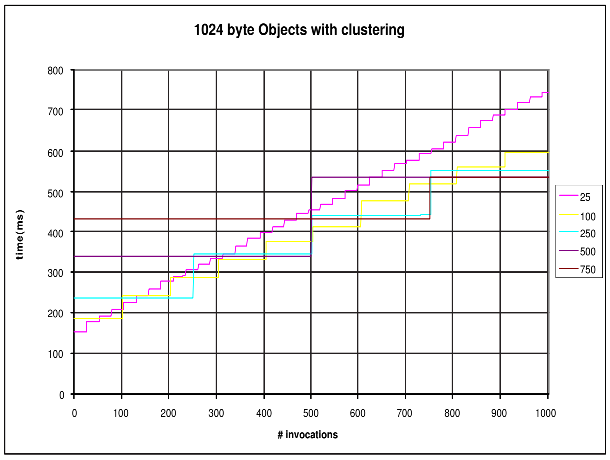
Figure II.4.3 shows the results for incremental replication with OBIWAN.Net. The two graphs at the top refer to incremental replication of 64 and 1024-byte objects, without clustering. They follow the same tendency than OBIWAN.Java. Nevertheless, the results are better in all aspects:



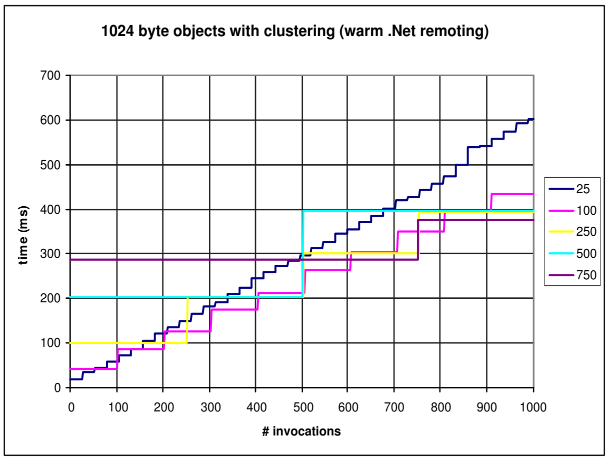
a. OBIWAN.Net 64-byte.



b. OBIWAN.Net 1024-byte.



c. OBIWAN.Net 1024-byte with clustering.



d. OBIWAN.Net 1024-byte with clustering and warm Remoting.

Figure II.4.3: OBIWAN.Net Performance Results.

- Incremental replication of one object at a time performs significantly better (a speed-up of almost 1.30, for 64-byte objects).
- The improvement is less noticeable when 25-100 objects are replicated each time (speed-up of 1.14 for 1024-byte objects).
- The performance is much better than with OBIWAN.Java when the number of objects being replicated is high (i.e., 250, 500, and 750).
- The curves in the graph (except for incremental replication of one object each time) are less separated.

These aspects can be explained by a number of reasons. Execution in OBIWAN.Net is faster, since code is just-in-time compiled, i.e., after the first invocation of a method, the code executed in future invocations is compiled code. This favors the creation of both replicas and proxies-in. Furthermore, the .Net CLR handles recursion better than the Java VM, since performance does not degrade, and it supports replication depths even larger than 750.

The graph of Figure II.4.3-c explore object replication with clustering, for medium-sized objects (1024-bytes). Comparing these with the results of OBIWAN.Java:

- In general, they are worse, with a slowdown ranging from 1.25 (for clusters of 250 objects) to 1.76 (for clusters of 25 objects).
- The results for clusters of 25 objects are significantly worse than those for larger cluster sizes.
- There is a much higher initial latency when replication starts, when compared to OBIWAN.Java. This difference was also observable without clustering but was not so significant.

These results indicate that when the burden of proxy-in creation is avoided, and despite execution in .Net being faster than Java, OBIWAN.Java performs better than OBIWAN.Net. This apparently indicates that communication with Remoting Services in .Net is less efficient than with Java RMI. In particular, it has a heavier initialization time, the first time a remote object is invoked (not present in RMI). However, this is due to the fact that, in .Net, a reference to a remote object can be created lazily, i.e., without actual communication, delaying it until the first actual invocation. Thus, this introduces an additional delay not measured in OBIWAN.Java

To mitigate the effects of this, we also measured the performance of object clustering in OBIWAN.Net with "warm" Remoting, in a number of tests . This is achieved by performing a prior, empty remote invocation on an object in P2, without any object replication taking place. This way, the first time that object replication takes place, the initialization delay of .Net Remoting is no longer reflected in the measurements. The results, except for clusters of 25 objects, become similar to those of OBIWAN.Java

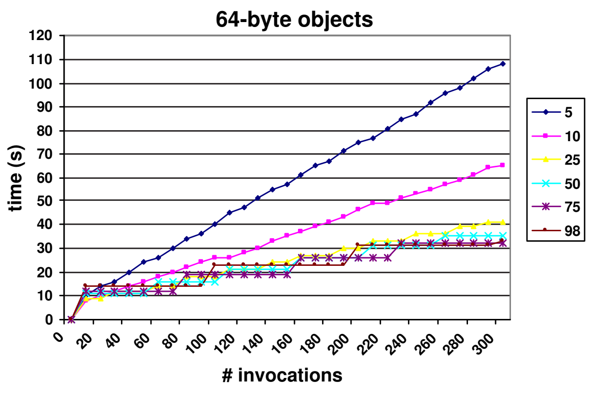
Summary The performance of incremental replication, without clustering, is much better with OBIWAN.Net than with OBIWAN.Java. Thus, in the case of applications where the programmer wants to refresh, update, or perform remote invocations on objects individually, it is the best option.

With object clustering, OBIWAN.Java outperforms OBIWAN.Net, without prior initialization of .Net Remoting Services. When this is done, the results become more similar, dominated by network transfer.

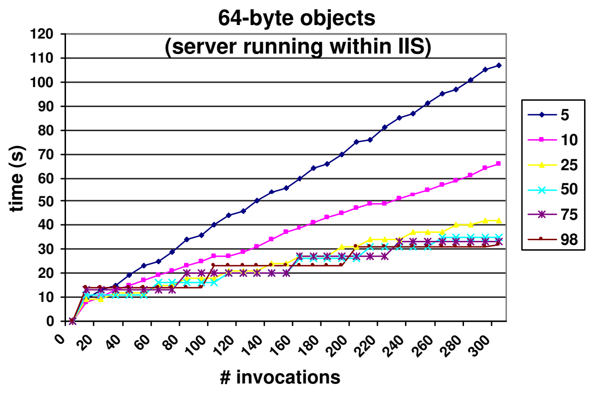
II.4.1.2 M-OBIWAN / OBI-Web / OBI-Per

In this section we present experimental results of incremental replication in the M-OBIWAN, OBI-Web, and OBI-Per prototypes. The performance tests were executed with the following infrastructure: a Pentium 4, 2.8 Ghz, 512 MB PC, and a IPAQ 3360 Pocket PC, equipped with 64 MB RAM.

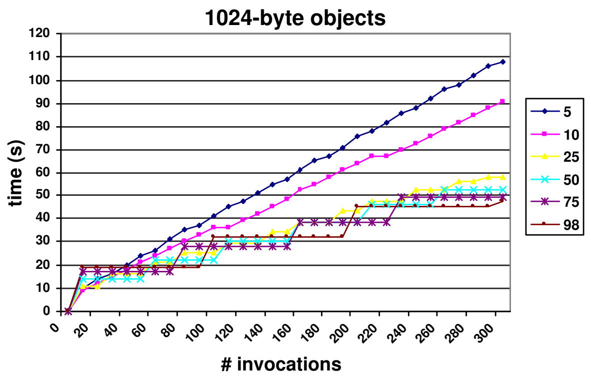
They were connected via wireless through a USB Bluetooth adapter at 700Kbps. This bandwidth value represents raw bandwidth. In .Net Compact Framework Framework, the communication between the PDA and the web-bridge is XML-formatted. Therefore, the useful bandwidth, when compared to a binary transport, represents only a fraction of this value.



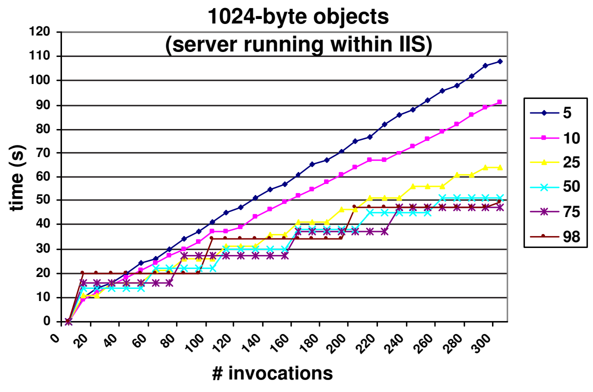
a. 64-byte objects.



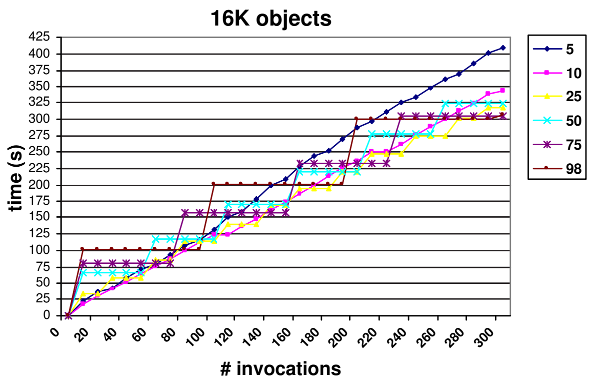
b. 64-byte objects integrated with IIS.



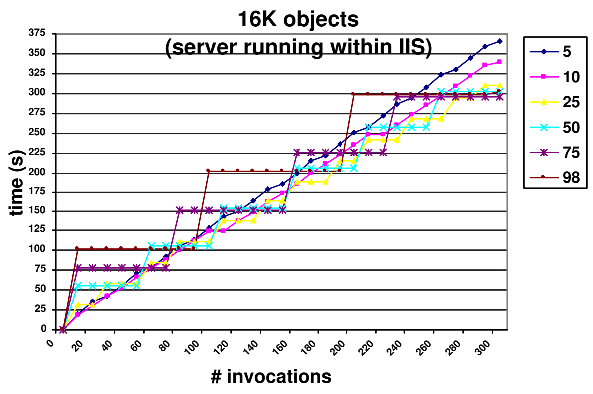
c. 1024-byte objects.



d. 1024-byte objects integrated with IIS.



e. 16 KB objects.



f. 16 KB objects integrated with IIS.

Figure II.4.4: M-OBIWAN/OBI-Web Performance Results.

We analyzed prototypes' performance with a micro-benchmark similar to the one used for the OBIWAN desktop prototypes. An application running on the PDA (process P1 acting as a client) performs a series of iterations executed on a list with 300 elements, stored in a desktop computer (process P2, acting as a server), with different payloads (64-byte, 1024-byte, and 16KB each), timing each invocation (see Figure II.4.4).

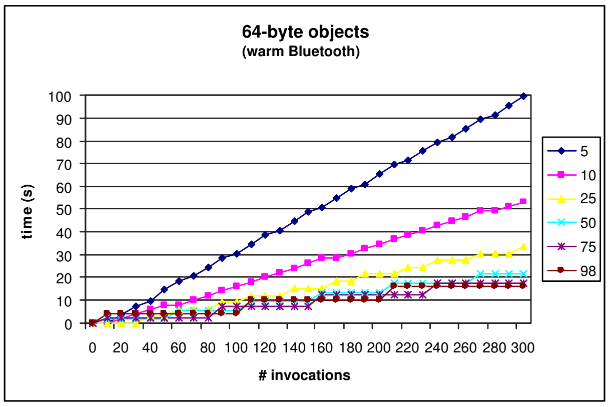
As the list of objects is iterated, on each object, a method is invoked. When the object is not yet replicated, the replication mechanism takes over and replicates the object where the fault occurred. Additionally, a configurable number of other objects is also replicated. In the end of each test, 300 objects have been replicated.

The same experiments were performed both with P2 running as a separate process, in M-OBIWAN; and inside IIS as a separate .Net Application Domain (`AppDomain`), in OBI-Web. In both cases, the communication between the web-bridge and P2 is performed via Remoting Services, with binary serialization. However, in the second case, an optimized version is used, since the web-bridge and P2 run in separate application domains, but sharing the same process. Finally, we evaluate the impact of adding persistence support, with OBI-Per.

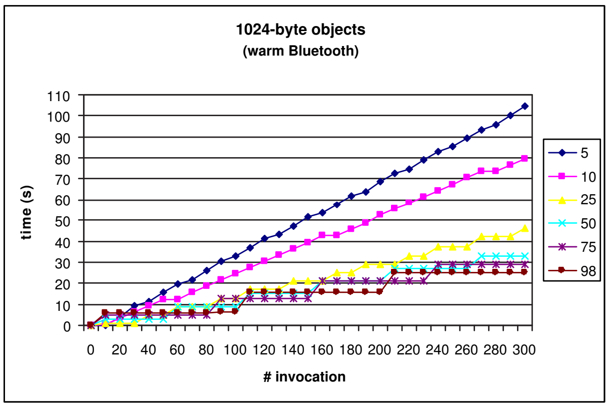
The replication mechanism was configured, by means of different policies, to replicate objects, on-demand, with a depth of 5, 10, 25, 50, 75 and 98 objects each time. This way, every time a proxy is replaced and the corresponding object is replicated, a number of others, referenced by it, are also replicated. The limit depth, 98, is imposed by stack restriction on .Net CF. The graphs of Figure II.4.4 show that:

- Replication performance is mostly latency-bound. Thus, the worst results are obtained with replication of 5 and 10 objects each time. It is specially noticeable with 64-byte objects. With larger payloads (16 KB), the effects of latency are naturally almost entirely masked.
- Incremental replication is more efficient when 25 or more objects are replicated each time. This is due to same reasons explained in the previous item.
- When object payload is raised from 64 to 1024 bytes (a sixteenfold increase), performance drops only by a factor of 1.6, on average.
- When object payload is raised from 1024 bytes to 16 KB (another sixteenfold increase), performance drops, roughly, by a factor of 7.
- The differences between M-OBIWAN (on the left) and OBI-Web (on the right) only become apparent when the object size is very large, for typical applications running in mobile constrained devices (16 KB). Executing P2 in an application domain within the IIS process, produces a speed-up that is always below 1.14.
- The first replication phase always takes more time than the following ones, throughout all graphs, even when very few (five) and small (64-byte) objects are replicated.

Naturally, incremental object replication of objects masks communication latency and minimizes memory usage by applications. The number of objects incrementally replicated each time, for near optimal results, needs not be too large (25 or 50). The best results are achieved with



a. 64-byte objects with Bluetooth warm network.



b. 1024-byte objects with Bluetooth warm network.

Figure II.4.5: M-OBIWAN/OBI-Web Performance Results with warm Bluetooth network.

higher replication depths (75 or 98). However, using these depths could waste more memory if only a few of the objects that are replicated, are actually accessed. Once more, XML-based serialization, imposed by .Net CF current limitations, is responsible for some wasted bandwidth[1] and increased processing-time due to parsing.

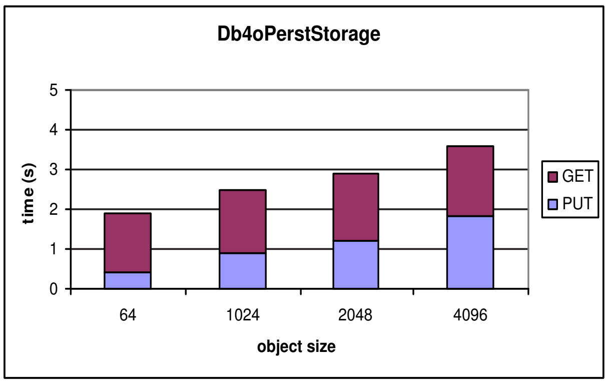
Finally, there is also a noticeable additional latency when the application starts replicating objects. To further investigate this, we repeated the experiments regarding M-OBIWAN (initially presented in Figure II.4.4-a and -c, with a slight modification. The results are portrayed in Figure II.4.5. In these experiments, the application initially performs replication of a small number of objects from a different list, that is not measured. Only after this initial replication is complete, does the the application starts iterating the test list. This way, object replication is performed using a "warm" (i.e., already completely established) Bluetooth connection. The differences observed correspond to a fixed-cost, i.e., the Bluetooth connection set-up time (around 8500 ms).

II.4.1.2.1 OBI-Per

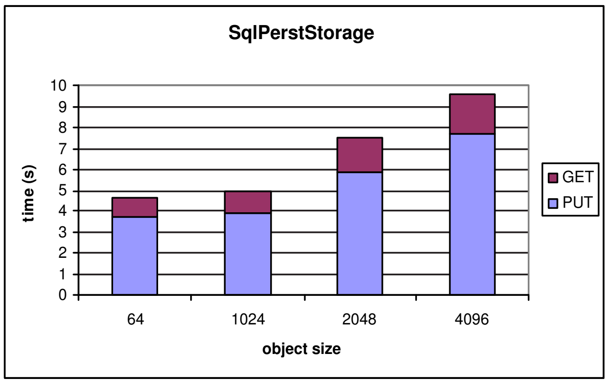
The evaluation of the introduction of persistence support in OBIWAN is divided in two parts. The first measures raw performance of the two persistence solutions used: i) SqlPerstStorage, using relational-based SQLServer; and ii) Db4oPerstStorage, using db4o, that is object-based. The second one measures the impact of the integration of persistence in overall performance, as perceived by applications. The results are presented in Figure II.4.6.

To evaluate the performance of the two persistence solutions, we performed a series of tests that consist in storing 1000 objects of different sizes (64, 1024, 2048, and 4096 bytes), and then retrieving them back. The connections to SQLServer and db4o are explicitly (re-)opened and closed, before and after each sequence of read or write operations.

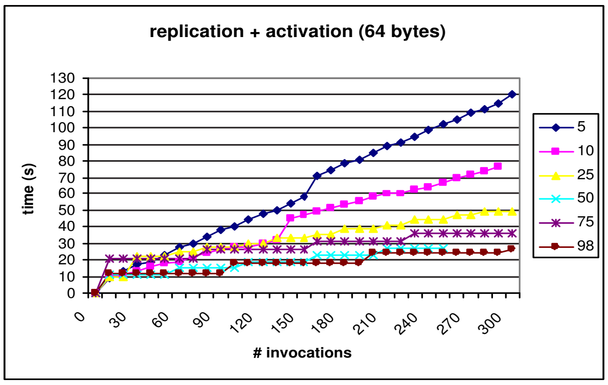
[1]When compared to binary serialization, the bandwidth required to serialize an object in XML is considerably higher.



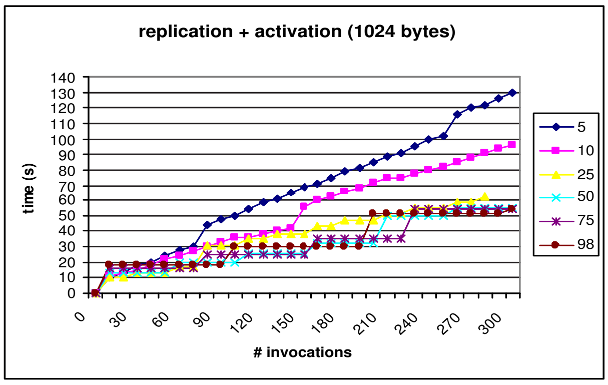
a. Performance of db4o persistence.



b. Performance of SQLServer persistence.



c. 64-byte objects with persistence.



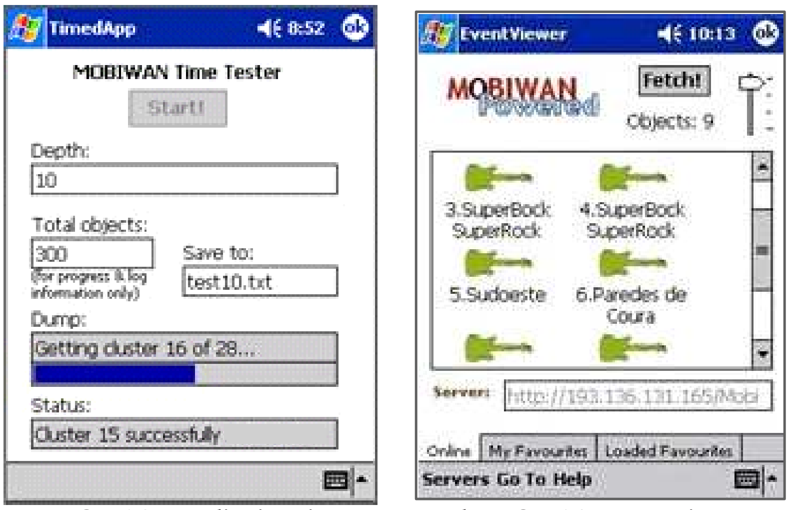
d. 1024-byte objects with persistence.

Figure II.4.6: OBI-Per performance.

The results shown in Figure II.4.6-a, and -b refer to total times after the two sequences (writing and reading), for each object size. W.r.t. reading objects from persistent storage, SqlPerstStorage performs better for smaller objects (especially for 64 and 1024-byte objects, with speed-up of 1.66 and 1.45). However, Db4oPerstStorage performs much better for writing objects, especially smaller ones. Speed-ups range from 4.27 to 9.25.

To evaluate the impact of introducing persistence support in common applications, we repeated the performance tests of M-OBIWAN and OBI-Web, now in OBI-Per, for objects sized 256 and 1024 bytes, using Db4oPerstStorage (see Figure II.4.6-c and -d). In these tests, P2 (that initially holds the objects and acts as a server) no longer loads the objects prior to application execution in P1. Instead, it only loads an initial object from persistence storage, and then the remaining objects are loaded only when their are needed for the first time. In this case, this happens when P1 tries to replicate another number of objects. All the experiments were performed with a ”warm” Bluetooth connection.

When objects are being replicated in P2, they are first loaded from persistent storage, and then method `get` of interface `IProvider` is invoked on them. The replication depths used are the same as before (5, 10, 25, 50, 75, and 98) and the same depth is used to load objects from persistent storage. As an example, using a replication of 25, method `get` is invoked only after those objects are loaded from storage. The most efficient approach would obviously be to perform



Depth: 10

Total objects: 300

(for progress & log information only)

Save to: test10.txt

Dump: Getting cluster 16 of 28...

Status: Cluster 15 successfully

Objects: 9

3.SuperBock SuperRock

4.SuperBock SuperRock

5.Sudoeste

6.Paredes de Coura

Server: http://193.136.131.165/Mobi

a. M-OBIWAN Replication Timer. **b. M-OBIWAN Event-viewer**

Figure II.4.7: M-OBIWAN / OBI-Web applications.

this in advance, as it would hide the resulting overhead. Even so, the results are not significantly worse. This is due to additional disk latency since the effect is more pronounced when the number of accesses is higher (using smaller replication depths), but it is mostly dominated by network latency and communication.

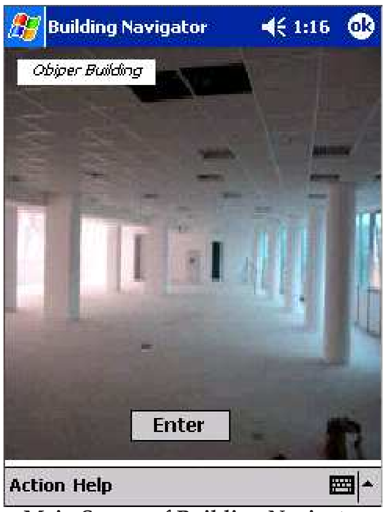
II.4.2 Example Applications

This section briefly presents a number of example applications developed using the OBIWAN implementations presented earlier. Regarding object replication, the amount of code the developer has to know about is restricted to OBIWAN API methods : i) GetObject once, in the beginning; and ii) PutObject, when sending updated objects to the server, e.g., before closing application.

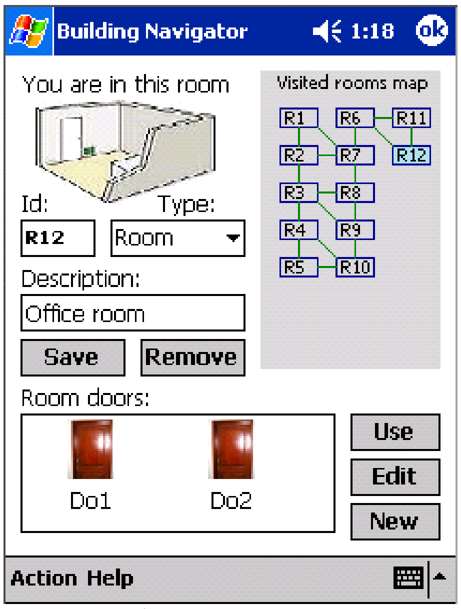
The initial example application, developed for a preliminary OBIWAN.Net prototype, consists of interactive tree-view explorer (Veiga 2002) that replicates nodes incrementally as their are visited by the user. It allows node creation, deletion, and modification of associated textual data. When instructed by the user, updated objects are transferred to the server. Figure II.4.7-a depicts a replication timing application used to obtain M-OBIWAN/OBI-Web/OBI-Per performance measurements, running on resource-constrained devices (e.g., PDA).

The **Event-Viewer** application, presented in Figure II.4.7-b runs on a PDA and is developed on top of OBI-Web, using `obicomp`. It allows users to create, share, and update schedules and additional information regarding cultural events. Data is organized as an object graph and stored in XML file. Users can determine the object server and the depth of the incremental replication mechanism. The list of events is replicated incrementally by user wish. Each event has further detailed information that is also replicated on demand, when a specific field is inspected.

Figure II.4.8 presents an simple **Interactive Building Creator and Navigator** developed on top of OBI-Per, that also runs on a PDA. The user is able to navigate and edit buildings previously constructed, or build new ones from scratch. Buildings are represented as object graphs

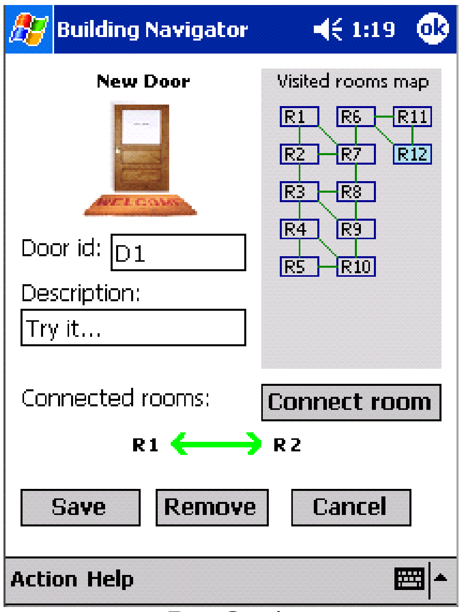


a.Main Screen of Building Navigator.



Id: R12
Type: Room
Description: Office room

b. Room Creation.



Door id: D1
Description: Try it...

c. Door Creation.

Figure II.4.8: OBI-Per Interactive Building Creator.

stored persistently. Users navigate among rooms, of different types, by following doors connecting them. A building is interactively created by alternate sequences of navigating through rooms, and creation of new rooms and doors. Doors are created by connecting two existing rooms. Rooms may have any number of doors. The user is informed of the path followed during a session. Users can freely modify an existing object graph by changing room and door details, delete doors and rooms, and change the pair of rooms connected by a given door.

Summary of Chapter: In this chapter, we presented the evaluation of the OBIWAN prototypes. The results obtained support the feasibility of the approaches followed. We started by studying the performance of incremental replication and object clustering in OBIWAN.Java and OBIWAN.Net, and then presented the performance results regarding M-OBIWAN and related prototypes.

We performed several experiments with OBIWAN.Java replicating objects of illustrative sizes (64-byte, 1024-byte and 16KB) incrementally, and with varying replication depths (25, 100, 250, 500, 750). We also measured the performance of replication with object clustering, using the same values for cluster sizes. The main findings were that: i) replication performs best with depths between 25 and 100, ii) replication depths of 500-750 are the least efficient, and iii) object clustering clearly outperforms incremental replication, by avoiding the creation of a large number of proxies-out and proxies-in.

Concerning OBIWAN.Java, we provided an additional analysis, by determining how replication times are split among relevant phases: i) replica creation, ii) serialization, iii) deserialization, and iv) network transfer, w.r.t. different replication depths and cluster size. This measurements further demonstrated the penalties associated with creation and serialization of proxy objects, when compared to just the creation of replicas.

The performance evaluation of OBIWAN.Net followed along the same line adopted for OBIWAN.Java, by varying object size, replication depth, and cluster size. They exhibit the same tendencies observed in OBIWAN.Java. When comparing the two prototypes, the main results, w.r.t. incremental replication were that in OBIWAN.Net: i) incremental replication has a 30% speed-up w.r.t. OBIWAN.Java, ii) incremental replication with optimum depths (25-100) offers a speed-up of 14% w.r.t. OBIWAN.Java, iii) as replication depths grow, the improvements over OBIWAN.Java become more noticeable due to more efficient handling of recursion. W.r.t clustering, the results in .Net were initially worse. Nonetheless, once lazy creation of remote references in .Net Remoting (as opposed to in Java RMI) was masked, the results became similar.

In the context of mobile environments, with wireless communication, we measured the performance of object replication in M-OBIWAN and OBI-Web. We followed an approach analogous to the one described earlier for desktop prototypes. In M-OBIWAN, replication depths were lower due to limited CPU, memory and bandwidth, always below 100. The results observed show the best replication depth to be 25, and that there is significant penalty imposed due to Bluetooth connection setup. The impact of providing support for object persistence in OBI-Per has been evaluated, and is not significant since total replication times were dominated by network transfer. Finally, we presented an overview of example applications developed on top of OBIWAN.

II.5 Conclusion

Part II of this dissertation was dedicated to Incremental Object Replication in OBIWAN. We proposed a portable way of performing object-fault handling, and incremental replication in distributed and mobile environments. It neither imposes changes to the virtual machine, nor the use of a specific or enhanced virtual machine. It runs on stock versions of the two most widely used virtual machines (Java and .Net).

We presented the related work concerning support for data-sharing. We addressed existing research projects and commercial technologies in the context of a proposed taxonomy. In the context of this taxonomy, OBIWAN is characterized as follows, in Table II.5.

Project	**System Architecture**	**Programming Model**	**Data-sharing Model**	**Propagation of Modifications**	**Portability**
OBIWAN	CS / P2P	objects	replication (also remote invocation and migration)	state-based	non-intrusive to VM code transparency (minimal API)

Table II.5.1: Design alternatives of OBIWAN, according to the taxonomy of Chapter II.1.

We presented in detail the architecture of transparent object-fault handling, and incremental object replication in OBIWAN. It is based on a set of OBIWAN core interfaces enabling these mechanisms, which are implemented by middleware code in proxies-out, proxies-in, and application objects, which are automatically generated. We further described object replication with variable depth and object clustering. All the mechanisms described are portable. Additional code to extend classes is automatically generated and hidden from the developer that does not have to know or edit it.

Applications manipulate replicated objects at full-speed, so there is no indirection neither when accessing object fields, nor when invoking object methods. When objects are not replicated, proxies occupy minimum storage, only the necessary to identify the corresponding object. Thus, proxies do not inherit from application classes, instead they implement a common interface. Therefore, there is no need for proxies to store as many fields as the actual objects. Even if they were null, this would waste memory (e.g., in a class with ten fields, the proxy would have to maintain ten null references).

There is no use of the reflection mechanisms of the underlying virtual machine during program execution, in order to access object fields or methods, since it would slow down applications considerably. In particular, transversing object graphs to create and update replicas is performed resorting to custom-purpose middleware code, which is automatically generated, and runs at full-speed; instead, transversing a graph using reflection mechanisms during run-time would slow down the replication process. Utilization of reflection is restricted to the class enhancement/extension phase performed by `obicomp` compilers.

OBIWAN has been implemented in a number of prototypes (OBIWAN.Java, OBIWAN.Net, M-OBIWAN) targeting the Java and .Net virtual machines in different environments. The prototypes execute on desktop and laptop computers in the context of wide-area distributed networks, and on resource-constrained devices (e.g., PDAs) used in mobile and pervasive computing. The limitations of existing virtual machines for mobile constrained devices (e.g., absence of remote method invocation and proper object serialization) have been circumvented by using a communication bridge based on web-services, and automatic conversion of objects into wrappers, using XML. OBIWAN has been integrated with application servers (OBI-Web) and persistence tools (OBI-Per). OBIWAN has been deployed within a commercial integrated development environment (e.g., Visual Studio), in order to fully automate application development.

The performance evaluation of the OBIWAN prototypes has been thorough and provided several results that were analyzed in depth. The tests consisted of several experiments with OBIWAN prototypes replicating objects of illustrative sizes incrementally, and with varying replication depths and cluster-sizes, in the context of both distributed and mobile environments. The performance results support the feasibility of the approaches described in this dissertation. Finally, the usability of OBIWAN has been evaluated by developing a number of example applications on top of it.

III

Automatic Memory Management of Distributed and Replicated Objects

(this page was intentionally left blank)

Soylent Green is made out of people!... – in "Soylent Green", Richard Fleischer, adapted from "Make Room! Make Room!", Harry Harrison

Part III addresses the automatic memory management of distributed and replicated objects, i.e., distributed garbage collection algorithms. After a brief introduction, we present related work regarding garbage collection, mainly concerning the distributed algorithms. Both complete and incomplete algorithms are presented. This stems from the fact that most of current complete algorithms are derived from the combination of efficient but incomplete ones, with special cycle detectors.

Following, we present three novel algorithms for complete distributed garbage collection. The first two refer to a distributed object scenario (both a centralized and a de-centralized cycle detection approach). The third one is an algorithm for complete garbage collection of replicated objects.

Then, the main aspects regarding the implementations are explained. The initial sections dedicated to implementation relate to application to object-oriented systems, while the latter ones concern application to web (HTML-based) systems. In both these types of systems, the notion of reference and reference graph is crucial, thus the relevance of the web scenario.

Both scenarios, with and without replication, are addressed, and performance measurements are presented. Part III closes with conclusions regarding the work developed w.r.t. automatic memory management.

(this page was intentionally left blank)

III1

Related Work on Distributed Garbage Collection

Automatic memory management, commonly known as garbage collection (GC), is a mature, yet challenging and dynamic research area whose early works date back to more than 40 years ago (McCarthy 1960; Collins 1960). Nonetheless, it is still a field of active and relevant research, also enjoying steadily increasing industry adoption in recent programming languages (e.g., Java, .Net and most object-oriented scripting languages), including distributed environments as in Java RMI (Wollrath et al. 1996) and and .Net Remoting (McLean et al. 2002).

It is widely recognized that manual memory management (explicit allocation and freeing of memory by the programmer) is extremely error-prone leading to: i) memory leaks, and ii) dangling references. Memory leaks consist on data that is unreachable to applications but still occupies memory, because its memory was not properly released. Memory leaks in servers and desktop computers are known to cause serious performance degradation. In addition, memory exhaustion arises if applications run for a reasonable amount of time (Willard and Frieder 1998; Willard and Frieder 2000).

Dangling references are references (e.g., pointers) to data whose memory has already been (erroneously) freed. Later, if an application tries to access such data, following the reference to it, it fails (i.e. referential integrity is not ensured). Such failure occurs because the data no longer exists or, even worse, the application accesses other data (that has replaced the one erroneously deleted) without knowing. Dangling references are well known to occur in centralized applications when manual memory management is used (Wilson 1992).

Besides providing programming soundness, modern GC offers performance benefits against manual memory management, and even against no memory management at all. GC contiguous allocation out-performs free-list allocation, and provides locality benefits since the other two options degrade object locality (Blackburn et al. 2004a).

Support for distributed cooperative work implies object sharing. The memory management of these distributed (and possibly persistent) objects is even more difficult to perform manually, and error-prone. Therefore, a number of distributed garbage collection (DGC) algorithms have been proposed in the literature and surveyed in (Plainfossé and Shapiro 1995; Jones 1999; Jones and Lins 1996; Abdullahi and Ringwood 1998; Shapiro et al. 2000; Ferreira and Veiga 2005).

In distributed systems, memory leaks in one computer may occur due to object references present in other computers. Furthermore, cyclic garbage complicates memory management even more, specially in distributed and/or persistent systems. Dangling references are also more common in a distributed environment. Such errors are harder to detect in distributed systems supporting replicated and/or persistent objects.

In summary, manual memory management leads not only to applications performance degradation and fatal errors but also to reduced programmer productivity. Thus, garbage collection, both centralized and distributed, is vital for programming productivity and system reliability.

Given that our contributions address distributed garbage collection (both for distributed and replicated systems), we address primarily previous work dealing with DGC. Nonetheless, we present a brief summary of local garbage collection techniques, i.e., GC for centralized systems, since some DGC algorithms are based on the adaptation of the former, to the distributed scenario. Then, we present DGC algorithms for distributed systems based only on remote method invocation (see Section III.1.2). Following, we present GC algorithms specific to systems with transactional and persistence semantics (see Section III.1.3). We continue with DGC algorithms for systems with data replication (see Section III.1.4), and we finish with some concluding remarks.

III.1.1 Local Garbage Collection (LGC)

In a centralized system, i.e., in the context of a single process, the only objects effectively available to an application are those that are reachable from some considered GC root-set, either directly, or indirectly, by transversing one or more references. Typically, GC root-set includes global[1] variables and thread stacks (i.e., local variables, method parameters).

Therefore, reachable objects are the only ones subject to being read or modified by the application. Thus, they are said to be *live*. All remaining objects are unreachable to the application and considered *dead*, i.e., garbage, and are only wasting memory and should be automatically discarded, by the garbage collector (Wilson 1992; Jones and Lins 1996).

In GC terms, an application is regarded as a *mutator*, an abstract entity with only one relevant operation: reference-assignment. This may result in: i) creation of a new reference to an object just created, ii) duplication of an already existing reference, or iii) discarding a reference to an object. Both i) and iii), or ii) and iii) may result from a single reference assignment. A side effect of iii) is that some objects may become unreachable.

A GC algorithm aims at three fundamental properties: i) safety, ii) liveness, and iii) completeness. An algorithm is safe when it does not reclaim reachable objects. Liveness, in GC, means that it eventually reclaims some garbage. An algorithm is considered complete if it eventually reclaims all garbage.

Garbage collectors can be broadly categorized in three main families: reference-counting (with several variations), tracing collectors, and hybrids of these two previous families (Plainfossé and Shapiro 1995).

[1]In object-oriented languages, these are `static` variables.

III.1.1.1 Reference Counting

The basic idea of the Reference-counting algorithm (Collins 1960; Collins 1961; Cohen and Trilling 1967; Cohen 1981; Christopher 1984) is the following: a counter is associated to each object denoting the number of references to it. When an object is created, a single reference points to it and its counter is accordingly set to one. Each time a reference is duplicated, the reference-count in the object is incremented. When a a reference to an object is discarded its reference-count is decremented. When its reference-count reaches zero, the object is no longer reachable and may be safely reclaimed by the local garbage collector.

This algorithm is mostly incremental, by nature, since its operations are interleaved with normal *mutator* action. However, object reclamation may, in turn, trigger decrements of reference-counts, with possible additional object reclamation, recursively. To limit collector latency, only a subset of newly-found unreachable objects should be considered each time, thus bounding pause time (Weizenbaum 1962; Glaser and Thompson 1987; Glaser 1987), and deferring reclamation. This maintains some *floating garbage*, i.e., garbage already identified but not yet reclaimed.

Reference-count maintenance imposes an additional overhead on every reference-assignment operation. It is observable that, for many objects, the net effect of a series of reference-count modifications is null (e.g., references from stack for parameter passing, and function return). Thus, it could be deferred (Deutsch and Bobrow 1976) (i.e., deferring identification) and become unnecessary later, with performance benefits (Baden 1983). Since the reference-counts may be temporarily inaccurate, special care is taken. Alternatively, all reference-count increments can be performed before any decrements (Bacon et al. 2001).

The storage overhead associated with reference-counts may be limited and, in the extreme, kept as a single-bit (Wise and Friedman 1977), since a majority of reference-counts does not surpass value one. Upon saturation of its reference-count (for any number of bits), the object will never be reclaimed, and another algorithm is needed to re-calculate reference-counts.

Reference-counting generates fragmentation, as free space recovered from reclaimed objects is interspersed with reachable objects. Reference-counting algorithms are not able, without specific adaptations, to identify and reclaim cyclic garbage (McBeth 1963). That is because, in cyclically referenced structures, an object may effectively be unreachable, yet its reference-count may still be greater than zero (e.g., a doubly-linked list).

Reference-counting algorithms are safe and live. They are not complete since they are unable to reclaim cyclic garbage.

III.1.1.2 Tracing

Tracing algorithms traverse the reference graph from the GC root-set, to identify which objects are reachable to the mutator. The objects that were not transversed are unreachable objects, and can be reclaimed. Tracing algorithms fall under two main sub-categories: i) mark-and-sweep, and ii) copying collectors.

Tracing collectors are safe, live and trivially able to reclaim cyclic garbage, i.e., they are complete. Therefore, most local garbage collectors are of this kind.

III.1.1.2.1 Mark-and-Sweep

A mark and sweep collector performs two phases (McCarthy 1960): i) a marking phase that traces the graph starting from the GC root-set marking every object found, and ii) a sweep phase that examines all the heap reclaiming unmarked objects.

During the mark phase, every reachable object is marked (e.g., setting a bit in the object header) and scanned for references. This phase terminates when there are no more reachable objects left to mark. During the sweep phase, the collector detects which objects were not marked and inserts their memory space in the free-list, for future re-allocation. Marked objects are unmarked to prepare the next collection. When this phase terminates, there is no more memory to be swept.

This algorithm introduces fragmentation, and the cost of object reclamation, during the sweep phase, is proportional to heap size. To avoid fragmentation, an extra compaction phase may be performed (Cohen and Trilling 1967; Carlsson et al. 1990). Objects that are found reachable after the marking-phase are moved together, in order to be contiguously grouped on one end of the object heap.

III.1.1.2.2 Copying Collectors

A copying collector also traces the object graph from the GC root-set but copies each object reached to another location (Fenichel and Yochelson 1969; Minsky 1963). Reclaimed memory is compacted which eliminates fragmentation. Heap compaction is thus performed *on-the-fly* w.r.t. tracing. Object allocation is performed by sequentially advancing a pointer.

The heap is divided in two disjoint semi-spaces called *from-space* and *to-space*. During mutator execution objects are allocated sequentially in *from-space*. Upon collection start, the collector moves reachable objects to the *to-space*. The objects left in *from-space* when every reachable object has been moved, are unreachable objects. The roles of the semi-spaces is swapped, i.e., flipped, atomically w.r.t. the mutator. The *to-space* becomes the *from-space* and vice-versa.

A clever implementation is due to (Cheney 1970) in which objects immediately accessible from the GC root-set are moved to *to-space* first. From this initial set, GC performs a breadth first traversal of the object graph, using two pointers (`free`, and `scan`). Every object in the *to-space* is scanned for references to objects still in the *from-space*. Each object reached is moved to the *to-space*, in the memory location pointed by `free`, that is updated accordingly.

All references in the scanned object are updated with the new locations of the moved objects. A forwarding pointer pointing to the location of the copied object in *to-space* is left in its old location, so that the algorithm remembers where the object has been copied to. When an object in *from-space* is to be re-scanned (via another reference), it is not copied again and the reference to it is simply patched, with the remembered value. Further optimizations are described in (Reingold 1973; Clark 1976).

A natural inconvenient of the copy algorithm is that only half of the memory space available is used at any point in time: the *to-space* is a wasted resource between collections.

III.1.1.2.3 Incremental Tracing

To reduce pause times, tracing and copying collectors can be made incremental, i.e., have their execution interleaved with the mutator (Steele 1975; Steele 1976; Dijkstra et al. 1978). The main issue w.r.t incremental tracing, or copying, is how to ensure the correct behavior of the collector when it and the mutator, interleaved, access and/or modify the same objects, before full collection is complete.

The problem with this interleaving lies in that while the collector is tracing the object graph, it may be modified as a result of mutator activity, without the collector knowing. This may lead to some reachable objects not being found by the collector.

One of the earliest solutions to this problem was the tricolor marking algorithm (Dijkstra et al. 1978), that provides an abstraction, applicable both to mark-and-sweep and copying algorithms. In this algorithm, objects are initially colored white. Coloring may be implemented by mark-bits, or by moving objects. Members of the GC root-set are conceptually black. Thus, when the collection is finished reachable objects are all colored black.

There is a third colour, gray, to signal an object that has been reached by the collector tracing, but some objects referenced by it, might not have been yet. When a gray object has been scanned it becomes black and its descendents are colored gray. In a copy collector the gray objects are those that have already been moved to *to-space* but have not yet been scanned. A collection terminates when there are no more gray objects to handle.

The invariant for correctness is the following: black objects may never reference white objects, since the latter will be wrongly considered unreachable. Yet, this could happen if a black object is modified after it has been scanned by the collector, and made to reference an object not yet scanned (a white object). If all the other references to the white object are destroyed, it will be incorrectly reclaimed. Therefore, either references to white objects, or modifications to black objects, must be tracked by the collector. This is achieved via synchronization, with the use of read and write barriers, respectively.

Read barriers are used in (Appel et al. 1988; Baker 1978; Brooks 1984; Queinnec et al. 1989; Zorn 1989; Huelsbergen and Larus 1993) to prevent the mutator from seeing white objects. In (Baker 1978), when a pointer to a white object is read, this action is trapped, and the object is colored gray (i.e., moved to *to-space*).

Write barriers are used in (Appel et al. 1988; Boehm et al. 1991; Demers et al. 1990; Dijkstra et al. 1978; Steele 1975; Yuasa 1990) when the mutator attempts to modify a black object. In (Nettles et al. 1992), the mutator always accesses and modifies objects in *from-space*, while the collector is copying reachable objects to *to-space*. When a black object (i.e., an object already copied and scanned) is further modified by the mutator, this is trapped by the write-barrier that logs and reapplies the modifications to the object in *to-space*, before flipping the spaces.

III.1.1.3 Hybrid Approaches

A number of algorithms implement an hybrid approach of tracing and some form of reference-counting (even when not stated as such). In essence, they apply tracing techniques only to subsets of the object graph, to improve responsiveness and/or avoid performing useless work. Techniques akin to reference-counting (e.g., remembered-sets) are used, complementary, to manage references among objects belonging to different subsets.

Since these solutions partition the object space in several subsets, and operate on them independently, these techniques provide a starting point for some GC algorithms targeting the distributed scenario (addressed later in this chapter). Examples of algorithms resulting from the hybridization of reference-counting and tracing include (Martinez et al. 1990; Lins 1992a; Lins 1992b).

III.1.1.3.1 Partitioned Tracing

With partitioned tracing, the whole memory is divided in several subsets called partitions. Each partition is traced independently from the others and cross-partition references are managed with a reference-counting algorithm (Bishop 1977; Moss 1989; Maheshwari and Liskov 1997c). The use of multiple independent partitions is very useful in large persistent (see Section III.1.3) and/or distributed (see Section III.1.2) systems, because each partition (or process) may be collected in parallel and independently. Small partitions are easy and fast to trace and allow earlier reclamation of intra-partition garbage. However, they introduce higher communication and storage overhead in managing inter-partition references.

III.1.1.3.2 Generational Collectors

Generational collectors are based on the empirical observation that newer objects have higher mortality (Lieberman and Hewitt 1983; Moon 1984; Ungar 1984; Ungar and Jackson 1988; Hayes 1991; Lins 1992b). Therefore, in many applications, most objects are reachable only for a very short period of time. Additionally, a fraction of the objects are very long-lived. Thus, generational collectors reduce the pause time due to GC, by reducing the amount of memory that has to be collected more frequently. Both mark and sweep and copy algorithms can be made generational.

This way, objects are discriminated in a number of partitions (at least two), that are called generations, managed accordingly to object age. Whence created, objects belong to generation zero, the youngest. As they survive collections, they are promoted to higher ranking, i.e., older generations. Younger generations are collected more often than older ones, since they are expected to contain more garbage objects.

References from objects belonging to older generations targeting objects in younger generations are kept in a *remembered-set*, that is included in the GC root-set of the younger generation. In order to construct *remembered-sets*, cross-generation references are detected using write-barriers or indirect pointers.

III.1.1.3.3 Train Algorithm

The Incremental Generational Collector proposed in (Hudson and Moss 1992) aims at reducing GC pause times even further. It is applied to the older generation, (i.e., *mature* objects that survived collection in the younger generation - the *nursery*).

The Mature Object Space (MOS) is divided in *cars* of fixed memory size. The collector reclaims at most one *car* each time it runs. *Cars* are grouped in *trains* with variable number of *cars*, and are totally ordered, since *trains* are ordered and *cars* are ordered (within a *train*). Objects surviving *nursery* collection are inserted in *cars* in any *train* that is not being collected. The algorithm moves objects closer and closer to the *cars* and *trains* they are referenced from. Intra-*car* garbage (cyclic or acyclic) is collected by tracing. This algorithm clusters related objects (referenced and referring) in the same *car* or, at least, in the same *train*.

Train collection is always performed on the first *train*. References to the first *train* are examined, in the GC root-set or from other *trains*. If there are none, the whole *train* is garbage and is reclaimed, otherwise the first *car* of first *train* is examined. *Trains* record references from objects in higher *trains* in their *remembered-set*.

References to the first *car* are examined as *cars* record references from objects in higher *cars* (including higher *trains*). If there are none, the whole *car* is garbage and is reclaimed; otherwise, objects are moved to the highest *car* they are referenced from. When there is not enough space in a *car*, each object is moved to a higher *car* or to a new one. If there are references from objects in the *nursery*, the object is moved to a later *train*.

When there are no more references to first *car* the whole *car* is garbage and may be reclaimed. Analogously, when there are no more references to any of the *cars* in a *train*, the whole *train* is garbage and all its *cars* may be reclaimed.

The functioning of this algorithm is graphically visualized and analyzed in (Printezis and Garthwaite 2002).

III.1.1.3.4 Ulterior Reference Counting

Ulterior Reference Counting (Blackburn and McKinley 2003) addresses the trade-offs between tracing and reference-counting. Tracing algorithms are complete and provide higher throughput via the generational approach, but impose higher maximum pause times due to full-heap collections. Reference-counting is incomplete and provides lower throughput due to costly pointer operations (partially addressed in (Deutsch and Bobrow 1976)) but offers higher responsiveness, without pauses due to full heap collections.

The two approaches are combined where they are best suited: i) generational tracing for newly allocated objects (*nursery*), that ignores reference modifications, copying surviving objects to mature space, and ii) deferred reference-counting for mature objects (mature space), generalized to heap objects (adding to registers and stack variables), with a variant of trial deletion to detect cycles in the mature space to ensures completeness.

Integration of the two spaces and the two algorithms is performed as follows. *Nursery* objects referenced by mature objects (*remembered-set*) are included in the tracing root-set. Mature objects referenced by *nursery* objects could be subject to a high number of reference-count increments/decrements. These are avoided by deferred reference-counting that also ”discards” modifications performed on short-lived objects.

III.1.1.4 Integration with Execution Environment

Reference-counting algorithms trivially need compiler support to insert specific instructions to update reference-counts when references are created, modified, and/or destroyed. Tracing algorithms must be capable of finding references from the GC root-set and inside each reachable object. The integration of a GC algorithm with the execution environment can be accomplished in a number of ways, described next.

If the execution environment provides runtime type information (e.g., Lisp, Smalltalk, Java, .Net), it is possible to differentiate references from raw-data (Wentworth 1990). When type information is not available (eg. C, and C++), solutions include: i) use of a pre-processor (Edelson 1992a), ii) compiler generating type information for each type (Boehm 1991; Ferreira 1991; Samples 1992), iii) take advantage of specific language features (Edelson 1992b; Detlefs 1992), like smart-pointers in C++, or iv) the collector is conservative (Boehm and Weiser 1988; Bartlett 1988; Boehm 1993) and regards any properly aligned bit pattern that could be the address of an object as an actual reference.

The last option may hinder completeness, wastes memory, increases pause times, prevents moving objects, with consequent fragmentation. Nonetheless, sometimes, it is the only feasible approach (Weiser et al. 1989).

Another conservative approach is presented in (Willard and Frieder 1998) with the global interception of manual memory management libraries to monitor allocated memory and periodically run the GC to identify memory leaks in log-running server applications.

Multiprocessor machines with shared memory use concurrent versions of algorithms, so that a number of processors may be executing application or mutator code, while other(s) are executing GC code. These algorithms are, originally, extensions of the incremental algorithms presented, with several threads running GC code. Recent work on GC for multiprocessor centralized systems includes (Domani et al. 2000; Levanoni and Petrank 2001; Lins 2002a; Azatchi and Petrank 2003; Lins 2005; Levanoni and Petrank 2006; Muthukumar and Janakiram 2006). The relevancy of these algorithms is expected to increase with the ongoing advent of multi-core processors.

The line of work presented in (Herlihy and Moss 1992; Detlefs et al. 2001; Detlefs et al. 2002; Sundell 2005) aims at developing garbage collection algorithms for systems with non-blocking (lock-free and wait-free) fairness guaranties to processes. However, they are not complete since they are exclusively based on reference-counting techniques.

The local garbage collector used in the the Java VM is generational. For younger generations (named *eden, survivor1, survivor2*), a copying collector, that compacts the object heap, is used. For the mature space, the older generation, Java uses a mark-and-sweep algorithm with compaction. Alternatively, it can be configured to use the train algorithm. The LGC included in the .Net CLR is a mark-compact hybrid that is generation-based, and manages three object generations.

For parallel machines, the Java VM uses one GC thread for each processor, to collect younger generations. This is optimized for very large heaps (e.g., in the order of several GB).

III.1.2 Distributed Garbage Collection (DGC)

Distributed garbage collection is a requirement for distributed object systems, that has been extensively addressed (Plainfossé and Shapiro 1995; Jones and Lins 1996; Abdullahi and Ringwood 1998; Shapiro et al. 2000; Ferreira and Veiga 2005). In these systems, distributed garbage, including distributed cycles, is frequent (Wilson 1996; Richer and Shapiro 2000).

In addition, when considering persistence, distributed garbage simply accumulates over time degrading performance. This is not simply an issue of disk space. Other aspects such as storage management, object loading on primary memory, serialization, etc. suffer performance degradations with the extra load imposed by the existence and increase of garbage.

In distributed object systems, there is a mutator, running on each process, performing reference assignments, either intra-process or inter-process. A distributed garbage collector is a set of components (one running in each process) that keeps track of inter-processes references.

The root-set of the distributed garbage collector (**GC root-set**) is the union of the local root-sets (**GC local-roots**) of all processes, that include global variables and thread stacks. *Live* objects are those that are reachable from the GC root-set, either directly or indirectly by transversing references in one or more processes, i.e., in one or more hops. The remaining objects are garbage since they cannot be accessed by the mutator running in any of the processes.

The distributed garbage collector is thus responsible for identifying *live* and *dead* objects in distributed systems. To do so, it manages reachability information (*local*, *remote*, and *global*) about objects. An object is reachable *locally* if it is reachable from the GC local-roots of its enclosing process. An object is reachable *remotely* if it is targeted by an inter-process (i.e., remote) reference. An object is reachable *globally* if it is live, i.e., if it is reachable either directly or indirectly, by transversing references in one or more processes, from a GC local-root.

Obviously, all objects reachable *locally* are also, trivially, reachable *globally*. However, the reverse is not true: an object may be be reachable *globally*, while being unreachable *locally*, because it is only referenced *remotely*, by *live* objects in other process(es). While *local* and *remote* reachability can be decided independently in each process, non-trivial *global* reachability cannot. Therefore, when there are remote references to an object in a process, that process can no longer decide locally whether that object is garbage. It must combine its local information with information provided by other (referring) processes.

Distributed garbage, as local garbage, may be acyclic or cyclic. Distributed cyclic garbage may, in turn, be comprised of connected subgraphs of cyclic and acyclic garbage, in any order, spanning several processes.

In distributed garbage collection algorithms, inter-process references are managed by auxiliary data structures. In the majority of the literature, it is established to name them stubs and scions. We recall the definition of the fundamental DGC data-structures (see Figure III.1.1) already presented in Part I.2. A scion (*entry-item*) represents an incoming reference, i.e., a reference pointing to an object in the scion's process. A stub (*exit-item*) represents an outgoing remote reference, i.e., a reference pointing to an object in another process.

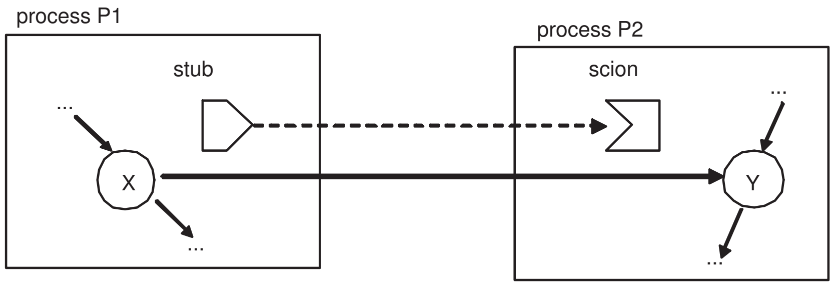


Figure III.1.1: DGC Structures: Stubs and Scions.

A distributed garbage collector should not manage intra-process references. Although both intra and inter-process references can be managed by the same collector, it has been determined that the best performance and scalability is obtained by employing an hybrid approach. In each process, there is a *dedicated* DGC component (usually reference-counting) to manage inter-process references; and a local garbage collector (Foster 1989; Dickman 1991; Juul and Jul 1992) (preferably, one that is complete, usually tracing) to manage references internal to the process. Thus, this hybrid approach is used throughout the literature, except for a number of exceptions (Mohamed-Ali 1984; Hughes 1985), since it is more practical.

Distributed garbage collectors, can be categorized in four main families, w.r.t. distributed garbage: i) reference-counting (with several variations), ii) tracing collectors, iii) hybrid approaches, and iv) specialized distributed cycle detectors.

Even though they are complete, tracing algorithms do not scale well to distributed systems as they traditionally impose greater disruption (synchronization and pause times) to applications in order to perform correctly. Therefore, most solutions rely on reference-counting algorithms for DGC due to scalability and performance reasons. Distributed reference-counting algorithms are scalable but do not reclaim cyclic garbage spanning several processes.

This raises the issue of completeness w.r.t. DGC, i.e., how to identify and reclaim distributed cycles of garbage. This has been often addressed with *combined* approaches, comprising an acyclic distributed garbage collector based on reference-counting, and a detector of distributed garbage cycles which is tracing-based.

III.1.2.1 Reference Counting

In this section we present distributed garbage collection algorithms that adapt the reference-counting technique, initially developed for local GC, to distributed systems. The algorithms are presented as successive efforts on avoiding the safety problems introduced by distribution, namely race-conditions between messages driving increment and decrement of reference-counts.

III.1.2.1.1 Distributed Reference Counting

A very simple adaptation of reference-counting to the distributed scenario was initially presented in (Nori 1979). Reference-counts are maintained, for each object targeted by remote references, at the process where the object resides.

Whenever a remote reference is created, duplicated or eliminated, an explicit message must be sent to the process holding the referred object. These messages result in the *increment* or *decrement* of the reference-count associated with the object.

This algorithm is highly vulnerable to communication anomalies (e.g., message loss and duplication). Replayed messages hinder algorithm completeness (w.r.t. increments) or safety (w.r.t. decrements). Lost messages prevent safety (w.r.t. increments) or completeness (w.r.t. decrements).

Even if messages are neither lost nor replayed, this algorithm is prone to distributed races between GC messages that prevent safety: i) *decrement/increment*, and ii) *increment/decrement*. These races occur when three processes are involved in reference creation and deletion: i) $P1$, *sender*, that holds a remote reference to object z (in $P3$), being duplicated to ii) $P2$, *receiver*, and iii) $P3$, *owner*, where object z resides.

Decrement/increment races occur when $P1$ duplicates its reference to z, sends it to $P2$, and then deletes its reference. If the *decrement* message from $P1$ arrives before the *increment* message that will be sent by $P2$, when it receives the reference, z will be prematurely reclaimed. *Increment/decrement* races take place when $P1$ duplicates its reference to z and sends it to $P2$, that immediately deletes it. If the *increment* message from $P1$, sent before sending the reference to $P2$, is delayed, it will arrive after the *decrement* message that was sent by $P2$. In this case, z is also wrongly reclaimed.

These problems are addressed in (Lermen and Maurer 1986) by using explicit acknowledgements messages for each increment/decrement message, before the next message is sent, thus introducing causality in communication among three nodes. However, this hinders algorithm performance and prevents its scalability.

The solutions presented next avoid the above mentioned races, while obviating the need for causality in communication, by avoiding the existence of competing messages, i.e., messages with causing antagonistic transitions (such as increment and decrement).

III.1.2.1.2 Weighted Reference Counting

An evolution to the previous algorithm is proposed in both in (Bevan 1987) and in (Watson and Watson 1987): it performs reference *accounting* without using counters. Instead, in this algorithm, each remote reference has an associated variable weight (*ref-weight*). The process holding the referred object ($P3$, *owner*) maintains information about the sum of *ref-weights* of all the references to the object, i.e., *total-weight*.

These weights obey to the following conditions. *Ref-weights* evolve monotonically (decreasing) and *total-weights* also evolve monotonically (decreasing from an initial *top-value*). Thus, there are no distributed races. *Top-value* is usually a positive number of the form 2^n.

When the first remote reference to an object is initially created (e.g., to $P1$), it is awarded *top-value* as its *ref-weight*. Accordingly, *total-weight* of the object, at $P3$, is also set to *top-value*. When a reference is duplicated to another process (e.g., $P2$), its associated *ref-weight* is split in half with the new remote reference. This involves no communication with the owner process $P3$ (as opposed to using an *increment* message). If $P2$ already holds a remote reference to the same object, it simply adds to it, the incoming *ref-weight*.

When a remote reference is deleted (e.g., in $P1$), a *decrement-weight* message is sent to $P3$, as the owner of the object, containing its *ref-weight*. $P3$ decreases *total-weight* associated to the object, by this value. Thus, *total-weight*, for each object, is always equal to the sum of all *ref-weights*. When *total-weight* reaches zero, the object is reclaimed.

There is a limit to reference duplication. When *ref-weight* of a reference reaches 1, the reference can not be duplicated further, i.e., an *underflow* occurs. This can be solved by using an extra indirection element to the remote reference, with *ref-weight* set to *top-value*, allowing further reference duplication. This situation introduces an indirection while invoking the object remotely; it is permanent and it may accumulate to a large number of references and objects (*domino* problem).

In this algorithm, reference elimination is still handled centrally (i.e., by sending messages to the owner process of the object). However, reference duplication is handled solely by the interaction of the two processes involved. No communication is needed to the process where the referred object resides. Thus, distributed races among reference duplication and elimination do not exist (since there are no increment messages). Nonetheless, it still is vulnerable to message loss and duplication.

As is the case with distributed reference-counting, Weighted Reference Counting is not able to reclaim distributed cycles of garbage. It can be combined with a tracing algorithm (Lins and Vasques 1991; Lins and Jones 1993; Jones and Lins 1992; Lins 2002b), when references are eliminated, to achieve completeness.

III.1.2.1.3 Generational Weighted Reference Counting

This algorithm (Goldberg 1989) introduces another approach to solve underflows in *ref-weights*. Each reference is represented by a *copy-counts* and a *generation*. Initial references belong to *generation* zero. When a reference is duplicated, its *copy-count* is increased and the new reference is assigned to the next generations. *Total-weight* associated with an object is extended to account for the number of references duplicated, per generation: *total-copies(n)*.

Eliminating a reference implies decrementing *total-copies(n)* of the corresponding generation, and incrementing *total-copies(n+1)* of the upper generation, by the exact number of times that the deleted reference has been duplicated. This can produce some negative values on some *total-copies* fields. Nevertheless, an object is reclaimed only when all *total-copies* fields are zero.

This algorithm avoids indirection at the expense of space overhead. Unfortunately, it is subject to overflow in *copy-counts*, though they occur with a much lower probability than the underflows of the previous algorithm.

III.1.2.1.4 Indirect Reference Counting

Indirect Reference Counting (Rudalics 1990; Piquer 1991) stores in each remote reference, the process where it was duplicated from (*parent*). Additionally, each reference holds a *copy-count* stating how many times it has been duplicated to other processes.

When a reference is duplicated, its *copy-count* is incremented. The newly created reference is initialized with a null *copy-count*, and registers the existing one as its *parent*. A succession of reference duplications where each newly created reference is further duplicated produces an inverted tree of indirections, whose root is the process where the remote reference was initially created.

When a reference is eliminated, the *copy-count* of its *parent* is decremented. If it reaches zero, that node of the inverted tree, may be reclaimed.

The main limitation of this algorithm lies in the fact that it only allows the deletion of references located at the leave nodes of the inverted tree of indirect references. This prevents reclaiming garbage at intermediary nodes (that constitutes floating garbage), where references no longer being used are maintained, until all the duplicated references in child nodes have been eliminated. This problem has been addressed by attempting to reorganize the inverted tree of indirections (Moreau 1998a; Moreau 2001). Another advance, ensuring algorithm correctness in the presence of object migration, has been addressed in (Piquer 1996).

Although providing a uniform solution to both underflow and overflow problems, and the distributed races presented earlier, this algorithm introduces storage overhead w.r.t. previous algorithms.

III.1.2.1.5 Reference Listing

Regardless of whether they are exempt from distributed races, all previous algorithms are still vulnerable to lost and duplicated messages. This is because messages are not idempotent. Thus, message idempotence is key for algorithm safety and completeness in the presence of communication failures, providing some degree of fault-tolerance. This is achieved with reference listing (Shapiro et al. 1990; Shapiro 1991a; Shapiro et al. 1992a; Shapiro et al. 1992b; Birrell et al. 1993a; Birrell et al. 1993b; Maheshwari 1994).

In Reference-listing, there are neither counters nor weights associated with remote references. Instead, each process stores, for each object targeted by remote references, an exhaustive list of the referring processes. This reference-list is kept free of duplicates since multiple remote references from the same process are represented by the same element in the list, regarding the referring process.

When a remote reference is created or duplicated to a process, the referring process must be inserted in the reference-list of the owner process. This equates to conceptually sending an *insertion* message.

Eliminating a remote reference implies that the (formerly) referring process must eventually be removed from the reference-list associated with the object. This equates to conceptually sending a *removal* message.

Insert and *remove* messages are idempotent as opposed to *increment* and *decrement* messages. Inserting an element in a list several times produces the same net result: the element is inserted in the list. The same is true for removals. On the other hand, repeating increment or decrement operations would corrupt reference-counts. By use of *insert* and *remove*, message duplication is handled safely. Nonetheless, correct ordering between *insert* and *remove* messages must be enforced to prevent possible premature object reclamation. This is achieved by numbering messages in sequence.

There are two main approaches to the actual implementation of reference-listing (SSP-Chains and Network Objects). We start by the latter that is acknowledgedly a simplification of the former in some ways. In Network Objects (Birrell et al. 1993a; Birrell et al. 1993b), there are explicit *insert* (*dirty*) and *remove* (*clean*) messages. Thus, for simplicity, when a reference is duplicated, the *receiver* (e.g., $P2$) process must send an explicit *insert* message to the *owner* process (e.g., $P3$). This message must arrive before any possible possible *remove* message from the *sender* process (e.g., $P1$), otherwise the object could be prematurely reclaimed. This needs that $P1$ maintains the object in $P3$ reachable until he receives acknowledgment from $P2$.

Within a process (e.g., $P1$), when all the objects containing remote references to a particular object, in another process (e.g., $P2$), become unreachable, this means that remote reference is no longer being used in $P1$. This is detected by the local GC in $P1$, and the corresponding *remove* message is sent to $P2$. *Remove* messages may be delayed and batched, but they must be explicitly acknowledged for liveness. Both kinds of messages carry sequence numbers to ensure message ordering.

In SSP-Chains (Stub-Scion Pairs) (Shapiro et al. 1990; Shapiro 1991a; Shapiro et al. 1992a; Shapiro et al. 1992b), remote references are represented by pairs of stubs and scions, possibly chained, that are created automatically when references are transmitted. Remote objects, nonetheless, are always invoked directly.

When a remote reference to an object is first created, a scion is created in the owner process (e.g., $P3$), and the corresponding stub is created in the referring process (e.g., $P1$). If $P1$ duplicates the reference to another process (e.g., $P2$), there is no need to contact $P3$. $P1$ simply creates a new scion that is chained to the existing stub. When $P2$ receives the reference, it creates the corresponding stub. Thus, the remote reference in $P2$ to an object in $P3$ is represented (w.r.t. DGC) by the chain of the two stub-scion pairs described. This eliminates delays in duplicating references among processes with the additional cost of indirections in DGC information. These, however, can be short-circuited, i.e., flattened lazily.

There are no explicit *remove* messages. Every time the local GC runs, it generates a new set of stubs, representing all remote references still in use by the process. From time to time, each process sends to other processes it contains references to, a special messages (*NewSetStubs*), with the list of its stubs corresponding to scions in that process. Deleted references are inferred when comparing stubs with their corresponding scions. Every scion without the corresponding stub may be safely eliminated. This avoids the need to re-send and acknowledge lost *delete* messages. *NewSetStubs* messages may be dropped, since only the most recent is required. To ensure ordering, *NewSetStubs* messages are also time-stamped.

III.1.2.1.6 Hierarchical Approaches to Distributed Reference Counting

The use of an hierarchical approach to DGC was initially presented in the context of distributed reference counting, with Group Reference Counting (Ichisuki and Yonezawa 1990). In this algorithm, the programmer must explicitly define object groups (also called regions or arenas), containing the objects where it is foreseeable that cycles will occur. It is thus of limited usefulness. An additional reference-count is maintained for the whole group, accounting solely for references from objects outside the group. When the reference-count of the group reaches zero, regardless of the reference-counts of each individual object (which may be non-zero, e.g., in a cycle), the GC knows it is safe to reclaim the whole group.

The same notion has been later applied to processes, with Hierarchical Distributed Reference Counting (Moreau 1998b). Processes are conceptually aggregated in domains (e.g., those running in the same cluster of computers) for DGC purposes. This algorithm is also able to shortcut chains of reference indirections. It also reduces the size of tables for reference listing (restricted to the number of processes within the same domain) and optimizes DGC message traffic.

III.1.2.1.7 Reference Counting with Information-Tuples

Reference-counts associated with objects are extended to reference-count vectors, called information-tuples, in (Philippsen 2000). Each element in the reference-count vector informs of how many times a reference to the object has been exported/imported to/from all other processes.

This algorithm avoids message races and acknowledgements at the expense of extra space for storage for reference-count vectors, and longer reference import/export messages (that contain information-tuples). Nonetheless, it provides low latency on garbage detection and uses less storage (since information-tuples are updated and combined, not recorded, as reference export/import messages are sent/received).However, since it is solely based on a generalization of reference-count, it is not complete, w.r.t. distributed cycles of garbage.

III.1.2.2 Tracing

In this section we present distributed garbage collection algorithms that adapt the tracing technique, initially developed for local GC, to distributed systems. The algorithms are presented as successive efforts on minimizing the need to block mutators, and synchronization among all participating processes when performing DGC.

III.1.2.2.1 Sequential Distributed Tracing

The straightforward adaptation of a tracing garbage collector for distributed systems was proposed by (Mohamed-Ali 1984). Mutators must be stopped at every process before the distributed marking phase can begin. At any point, any process may decide to initiate a collection (e.g., because it is almost out of memory). This decision must be transmitted to a coordinator process. This role may be statically assigned or decided dynamically, in this case, requiring a distributed consensus.

Upon notification of the request, the coordinator process instructs all mutators to stop, and initiate the marking phase. Marks are propagated from GC local-roots along both intra and inter-process references. References are transversed and objects marked using a depth-first approach in order to minimize memory usage.

Once the marking phase is terminated in all processes, and there are no more marking messages in transit, the sweep phase can be performed independently by each process.

III.1.2.2.2 Concurrent Distributed Tracing

Stopping the mutator of every process and requiring that all processes perform DGC simultaneously imposes considerable performance penalties, application disruption, and limits scalability to large numbers of processes.

The algorithm proposed in (Hudak and Keller 1982) is a distributed version of the tri-color incremental garbage collection (Dijkstra et al. 1978), with the additional constraint of existing a single root for the whole distributed object graph (which is suited to the functional programming environment that the algorithm addresses).

This adaptation consists of transforming each recursive marking step in an autonomous task executed concurrently with the others. For this, each process keeps a list of active mutator tasks and another list with GC tasks. GC tasks terminate only after the termination of all the tasks spawned from it. Mark phase termination implies distributed consensus, and the sweep phase can start only after termination of marking phase in all processes.

The meaning of the three colors used (*white, grey, black*) is also adapted to the distributed scenario. In this case, the color *gray* is used to identify already marked objects but whose recursive tasks spawned from it have not finished yet. This algorithm still demands global synchronization between the two phases, and the high number of marking tasks spawned creates high network traffic and processor load .

III.1.2.2.3 Mark-and-Sweep with Time-stamps

Global propagation of time-stamps until a global minimum can be computed was first proposed in (Hughes 1985) to detect distributed cycles of garbage. This algorithm proposes a generalization of marks to accommodate numerical time-stamps, instead of just a small fixed set of options (e.g., reachability-bits, colors).

In this algorithm, local GC in each process propagates time-stamps from scions to stubs, following local reachability paths. Before a local collection, each GC local-root is marked with the current time. GC local-roots and scions are traced in decreasing order of time-stamps, which ensures that each stub may be marked only once in each local collection, while retaining the highest time-stamp of the roots and scions that lead to it.

Thus, the marks associated with live objects are increased, while the ones of garbage objects remain constant, once they become unreachable. The DGC component propagates marks from stubs to their corresponding scions in other processes, to achieve the same result globally. Upon reception of a marking message (bearing updated time-stamp of a stub), the corresponding scion time-stamp is updated if the value received is greater than its current time-stamp. If updated, the old scion time-stamp is used to possibly update the *redo*[2] value of the process, noting the fact that this newly received time-stamp has not been propagated locally yet.

When the receiver acknowledges reception of all messages, the sender can update, i.e. increase, its *redo* value to the largest time-stamp it has propagated. If a global minimum for *redo* values in all processes can be computed, this can establish a threshold to determine if a given object is garbage. This is because, in such conditions, its time-stamp will not possibly increase in the future.

This algorithm is complete and allows concurrent marking in all processes, requiring synchronization only when deciding on a new global minimum *redo*. Nonetheless, it is not scalable for several reasons. Determining the minimum *redo* value depends on a global termination algorithm (Rana 1983). The clocks in processes must be synchronized, message latency bounded, and it needs the cooperation of all processes in order to progress. If even just a single one of them does not, the value of global minimum will effectively freeze, and no new garbage can thus be detected from that moment on, even if that process does not contain garbage.

III.1.2.2.4 Logically Centralized Tracing

The algorithm presented in (Liskov and Ladin 1986) introduces a centralized approach to DGC, in which distributed garbage collection is performed by a logically centralized server, receiving graph information from all processes. The centralized server performs complete distributed garbage collection and informs processes of objects (i.e., scions) to delete. This garbage collection service, though logically centralized, may be physically replicated for increased availability and performance. Replicas exchange information via ”gossip” (Fischer and Michael 1982) messages.

Each process performs asynchronous local garbage collection (LGC) and computes accessibility information of scions (*inlists*) and stubs (encoded in locally known paths), as well as references in transit. The central service continuously collects accessibility information from each node and maintains a view of the global graph. When requested, it informs each process of which scions it should delete, since there are no longer any stubs pointing to it. The view of the global graph is inconsistent but follows a conservative approach to ensure safety w.r.t. garbage collection. All messages must be time-stamped, should have bounded latency, and process clocks must be loosely synchronized.

This algorithm does not require global synchronization of nodes, as there are no global distributed phases, nor it needs all processes to participate continuously. To detect distributed cycles of garbage the central service may require receiving several messages from each process, with added delay, and communication/processing load. Thus, it can suffer from congestion

[2]The *redo* value in each process stores the maximum value received in messages and already propagated locally.

because it performs all DGC tasks, and not only those related to cycle detection. Therefore, reclamation of acyclic distributed garbage may also be delayed unnecessarily.

The initial version of this algorithm has some problems w.r.t. safety, described in (Rudalics 1990). These problems were later addressed in a revised version of the algorithm described in (Ladin and Liskov 1992). It uses the technique proposed in (Hughes 1985), and described in Section III.1.2.2.3, but in this case the time-stamp minimum is calculated centrally.

Requirements of clock synchronization, bounded message latency, and the fact that the central server performs all the work w.r.t. DGC, make this algorithm unscalable to wide-area scenarios. In these scenarios, it may be unpractical to enforce clock synchronization, it is very difficult to limit message latency, and dependence on a central server will slow down GC in most processes.

III.1.2.3 Garbage Collection of Distributed Cycles of Garbage

There are two types of distributed garbage collection algorithms to detect and reclaim distributed cycles of garbage objects: i) full-scope, and ii) suspect-based detectors. Full-scope detectors detect all existing cycles simultaneously. They impose additional work globally to the system, even when and where there are no cycles to detect. Continuous extra-cost (in terms of time, space, and messages) leads to wasted work. All tracing algorithms are, by nature, of this type.

Suspect-based detectors (or per-cycle detectors) try to identify cyclic garbage, starting with an object suspect of belonging to distributed garbage. In each detection, they identify at most one distributed cycle of garbage, instead of all at once, as full-scope algorithms do. Heuristics are needed to select a candidate object (e.g., distance from GC root-set, local un-reachability). The cycle detector confirms if the object actually belongs to a distributed cycle. Several cyclic graph detections may run concurrently. These algorithms impose specific work for each cycle detection but restricted only to processes related with cycle being detected.

We will describe algorithms of both types. Suspect-based detectors include those that use: i) trial deletion, ii) object migration, iii) back-tracing, and iv) group merger. Full-scope algorithms include those that employ: i) group tracing, ii) monitoring mutator events, iii) distributed train, and iv) mark propagation with optimistic back-tracing. The algorithms are presented following a quasi-chronological order. This helps the description since there are algorithms of different types that employ techniques that are related (e.g., group tracing and group merger, back-tracing and optimistic back-tracing), while providing improvements over previous work.

III.1.2.3.1 Trial Deletion

The work presented in (Vestal 1987) proposes trial deletion to detect distributed cyclic garbage. Besides reference-counts for each object, it maintains an additional reference-count, specific for trial deletion, in each object. These reference-count fields are used to propagate the hypothetical effect of trial (simulated) deletions. Trial deletion starts on an object suspect of belonging to a distributed cycle, heuristically chosen. The algorithm then simulates the recursive deletion of the candidate object and all referenced objects. When, and if the trial counts of every object of the sub-graph drop to zero, a distributed cycle has been successfully found. It may then be safely deleted as it is certain that there are no other references from objects outside the suspected cycle.

This algorithm imposes the use of reference counting for LGC (which is seldom chosen); this is an important limitation. The recursive freeing process is unbounded, since the size of a cycle is not anticipated. Poor candidate election (e.g., a live object) will lead to wasted trial deletions of a large number of objects in many processes. Furthermore, it is unable to detect mutually referencing distributed cycles of garbage.

III.1.2.3.2 Object Migration

Migrating objects to a single process in order to convert a distributed cycle into a local one, that is traceable by a basic LGC, has been used or otherwise suggested by several authors (Bishop 1977; Shapiro et al. 1990; Maheshwari and Liskov 1995; Piquer 1996; Maheshwari and Liskov 1997b). The main task of a migration-based DGC algorithm is cycle consolidation, i.e., co-locate all objects belonging to a distributed garbage cycle in a single process.

In (Bishop 1977), cycle consolidation is initiated with a suspect object chosen heuristically (e.g., locally unreachable, long time without invocations, increase in reference deletion messages received by the process since the object was created or last used). Then, every object belonging to the hypothetical cycle is migrated to one of its referring processes. Once the distributed cycle is confined to a single process, any tracing algorithm is able to reclaim it. Reference-counting extended with cycle consolidation ensures completeness.

Globally, object migration, for the sole purpose of DGC, is a heavy requirement for a system, needs extra and possible lengthy messages (bearing the actual objects) among participating processes, and all processes must adhere to the same homogeneous architecture. It may take several iterations to consolidate a single cycle. It is very difficult to accurately select the appropriate process that will contain the entire cycle. Additionally, cycles that contain many objects, if copied into a single process in charge of tracing them, may cause it to overload. Furthermore, heuristics to decide object migration and destination process are not optimal. Thus, some live objects may be migrated alongside with garbage ones, and it is difficult to select the process to receive the entire cycle, while minimizing the cost of object migrations. Some of these limitations have been addressed differently in following work, presented next.

The work presented in (Shapiro et al. 1990) introduces two optimizations. It restricts the directions of objects migration using a total order among nodes, thus ensuring all objects in a cycle will converge to the same process, i.e., the upper-bound process (e.g., with the highest process identifier) in that subset of processes. This prevents a single process from being overloaded with all cycles, though higher-ranking processes will often have higher load. Objects are only migrated *virtually*, in the sense that they are not physically transferred among processes, but simply marked as belonging to a different logical space. However, tracing of logical spaces that are now spread among processes will incur in additional network traffic.

An alternative approach that allows the use of reference-counting algorithms is proposed in (Gupta and Fuchs 1993); a fixed process is designated as the dump, where all objects belonging to cycles are migrated to. It obviates the need to know referring processes explicitly, but introduces a single-point of load and failure (the dump). Some live objects are still migrated to the dump unnecessarily.

III.1.2.3.2.1 Controlled Object Migration A more efficient migration heuristic based on distance from GC local-roots is introduced in (Maheshwari and Liskov 1995).[3] The distance of an object is the minimum number of inter-process references from a GC local-root leading to the object. GC local-roots have zero distance and new scions have initial distance of one. The LGC propagates increased (unitary increments) distances from scions to stubs. Scions are traced in increasing order, and since each object is traced once, only the minimum distances will be propagated. Thus, the distance of garbage objects grows unbounded.

Once the distance exceeds a pre-determined threshold (T), the object is eligible for migration. Small values of T may cause unnecessary migrations, while large values will delay cycle detection. A small multiple of the expected maximum distance should be used. An object to be migrated is batched along with other objects referenced by it, and with greater distance values, as they are likely to be garbage as well.

The selection of the destination process is also heuristic and based on hints that are propagated along with distance values. Initially, they contain the processes of stubs reachable via suspected objects. Only the highest process identifiers are propagated, thus ensuring convergence of the selection process, similarly to (Shapiro et al. 1990). This minimizes migration of objects to different processes, as it transfers groups of objects in a single message, but it introduces additional complexity with propagation of distances and hints.

III.1.2.3.3 Group Tracing

Group Tracing proposes an hierarchical approach to DGC based on a tracing algorithm. Distributed tracing is performed within groups of processes (Lang et al. 1992), that may be created and managed dynamically and belong themselves to other higher-level groups. Groups may have any number of processes and may intersect, thus sharing processes. Tracing is initiated only when LGC cannot reclaim enough memory.

In this algorithm, group tracing is performed on top of a distributed reference-counting mechanism, since scions keep reference-counts. The LGC at each process propagates reachability marks locally from scions to the stubs they reference. There are two sequential tracing phases in each process, one named *hard* and another named *soft*. *Hard* marks denote objects that are referenced from outside the group (conservatively regarded as roots), or from GC local-roots within the group. *Soft* reachability marks mean the object is referenced from within the group.

DGC propagates hard marks from stubs to the corresponding scions in other processes in the group. This phase terminates when the group is stable, i.e., where there are no more messages

[3]This heuristic was also independently presented one month later in (Fuchs 1995), for an algorithm with a different approach, but it is not often considered.

in transit, with *hard* marks, and there is no data (regarding marks) being produced in any of the group processes. This is hard to establish and requires a form of distributed consensus. After stabilization, every scion marked *soft* is neither reachable from outside the group nor from any GC local-root in any of the processes of the group. Therefore it may be explicitly deleted to break the distributed cycle. Distributed reference counting will reclaim the garbage, since it is now acyclic.

This algorithm does not require every process to participate in cycle detection, as opposed to (Hughes 1985) and failure of processes is handled by the dynamism of group creation. A new group is formed with the surviving processes that allows detection to continue. This approach is scalable to large numbers of small groups, but not to higher-level groups spanning many nodes, since they require a distributed consensus by the participating processes on the termination of the global trace and every object is marked individually. Questions regarding the safety of this algorithm, w.r.t. concurrency between mutator activity and cycle detection, have been raised in (Maheshwari and Liskov 1997a).

III.1.2.3.4 Distributed Back-tracing

Distributed Back-tracing was initially proposed in (Fuchs 1995), but not implemented. This initial approach does not provide a mechanism for calculating back-tracing information (i.e., *inverse reference graph*), which is assumed to already exist.

The work in (Maheshwari and Liskov 1997a) follows the same approach, while presenting an implementation, and an optimized mechanism to calculate back-tracing information. Distributed back-tracing starts from suspected objects (of belonging to a distributed cycle of garbage using the heuristic devised in (Maheshwari and Liskov 1995)), marking objects, until it finds GC local-roots (aborting cycle detection) or when all objects leading to the suspect have already been back-traced, meaning that a distributed garbage cycle has been correctly detected.

There are two mutually recursive procedures available in each process: one to perform local back-tracing and another one is in charge of remote back-tracing. Local back-tracing is performed on objects holding outgoing remote references (i.e., those referencing stubs). Each of them is appended with a *leader* field, indicating other local objects that directly or indirectly reference it, and that are targeted by incoming remote references (i.e., with associated scions). Stubs and scions are collectively regarded as *iorefs*. *Leader* fields are calculated as a result of local tracing. Thus, individual objects need not have explicit backward references as in the model proposed in (Fuchs 1995).

Distributed back-tracing is performed by remote invocation to propagate marks along *iorefs* of the processes containing the suspected cycle. Distributed back-tracing thus results in a direct acyclic chaining of recursive remote procedure calls. To ensure it is acyclic (i.e., to ensure termination and avoid looping during back-tracing that spans several processes), each *ioref* must be marked with a list of *trace-ids* to remember which back-traces have already visited it. This also ensures safety in the presence of multiple overlapping detections.

To ensure safety w.r.t. the mutator, reference duplication (both local and remote) must be subject to a transfer-barrier that updates *iorefs*. The distributed transfer barrier may need to send extra messages that must be guarded against delayed delivery.

Chaining of remote procedure invocations along a possibly large number of processes, and the need to keep state in processes about detections in course, hinders scalability and fault-tolerance.

Distributed back-tracing is also used in (Rodriguez-Rivera and Russo 1997) for cycle detection in CORBA (Siegel 1996), addressing detailed implementation issues, on production systems composed of commercial-off-the-shelf (COTS) software. This work introduces optimizations such as: i) generational back-tracing (that consists in back-tracing from more recent objects first, and more often than from mature objects), and ii) back-trace re-factoring, that reduces the number of redundant back-traces, namely of objects along the same reference path of others that triggered previous back-traces.

III.1.2.3.5 Monitoring Mutator Events

Relevant mutator events (i.e., reference creation and destruction), and causal dependencies among them, are monitored in (Louboutin and Cahill 1997) to perform DGC. It introduces a lazy approach to DGC, in contrast with other algorithms that operate by eagerly registering creation of inter-process references, through control messages. In this algorithm, all messages are delayed until they become necessary.

Only reference creation and destruction involving remote references are relevant. Objects targeted by remote references are designated global-roots. Analysis of mutator events is used as an alternative to tracing the distributed object graph. Global-roots conceptually exchange messages, one for each relevant event that equates to reference creation and destruction. These events are recorded in each global-root but the corresponding messages are not sent immediately. Thus, multiple messages regarding creation and destruction of references may be batched and sent lazily.

Each event is time-stamped with a respective direct dependency vector (DDV) that reflects causal dependencies on operations performed by other global-roots. DDVs are logged, merged with its causal predecessors, and propagated, until full vector-time is obtained for each global-root. This enables calculation of the complete transitive closure of the graph. A complete causal-cut of reference creations and destructions allows identification of distributed garbage (both acyclic and cyclic).

This algorithm is complete. It is resilient to message loss and duplication, and lazy message exchange avoids races and synchronization bottlenecks. However, it has unbounded latency for all garbage detection (not just for cyclic garbage) and increased space overhead to store message logs. The use of individual vector clock logs, for each global-root (each object referenced remotely), further exacerbates storage overhead.

III.1.2.3.6 Distributed Train Algorithm

An adaptation of the Train Algorithm to distributed scenarios is proposed in (Hudson et al. 1997). Objects reside in cars of fixed size and each car resides on a single node. Trains comprise several cars possibly spanning several processes. This allows detection of inter-process cyclic garbage. Analogously to the centralized scenario, each process must have more than one train. Cars, trains, and processes, must form a completely ordered set. The algorithm may be used with or without object migration.

A pointer tracking mechanism in each process monitors events of creation, deletion and transfer of references, and communicates them to the process involved (i.e., *sender, receiver* of references and object *owner*). Events are optimized using several techniques: i) redundant creation and deletion messages are omitted, sending only those that may change object state from unreachable to reachable (and vice-versa) in a process, ii) events are piggy-backed on other messages (e.g., remote invocations), iii) event compression in messages, and iv) event combination in processes.

Each process maintains two tables with the cars that have references to its objects, that function as *remembered-sets*: i) an up-to-date version, and ii) the *sticky* version. The up-to-date version is a subset of the *sticky* version. The *sticky* version, for completeness, records all cars that were ever known to the process, along with a *changed-bit*. Each train has a master, the process where it was created, that manages and eventually cleans up the train. The processes holding cars of a train are linked in a logical token-passing ring.

The token is always held by one process of the ring, initially by the master process. Each process receiving the token (e.g., $P1$ $\{P1\}$) either holds, or relays it as follows:

- i) if $P1$ has external references in the train sticky-remembered-set, it must hold the token until it has none. Then it must re-start token with value $\{P1\}$.
- ii) If $P1$ has no external references in train *sticky-remembered-set* but *changed-bit* is true, it re-starts token with value $\{P1\}$ and resets *changed-bit*.
- iii) if $P1$ has no external references in train *sticky-remembered-set* and *changed-bit* is false, it relays the token unchanged.

The token will make, unchanged, at most two rounds of the ring, the first one to reset *changed-bit* at every process, and second one to accomplish garbage detection. In Rule iii), if token value is $\{P1\}$ already, the whole train is found to be garbage and can be reclaimed, since the token took a complete circle of the ring, with empty *sticky-remembered-sets* and all *changed-bits* reset. This acts like a distributed termination algorithm. To allow concurrency with the mutator, w.r.t. inserting new cars in existing trains, cars in trains are always kept in two separate sets (old and new *epochs*). Garbage detection is restricted to the old *epoch.* When this is eventually reclaimed, new epoch flips and becomes the old epoch, and the new epoch is emptied.

This algorithm is complete and regarded as incremental, highly asynchronous, and mostly concurrent with the mutator. However, the adaptation to distributed scenario introduces high complexity and train management requires that processes maintain state about garbage detections in course, with additional complexity to account for cars joining/leaving the train while detection algorithm is running. Moving cars (and possibly objects) causes increased inter-process messaging, due to the fact that a single car, until being reclaimed, can be moved among several trains, which imposes delay on garbage detection. Moreover, cyclic garbage may delay prompt detection of acyclic garbage if they reside in the same train. Trains may become very large, which is unimportant for local GC, but becomes an important issue important in large scale distributed scenario with long rings of processes managing trains.

In the unfavorable example where one train may contain distributed garbage cycles and many other acyclic garbage elements, the detection of any may imply/depend on the detection of all others. Thus, while eventual co-location of all cyclic garbage is guaranteed eventually, it may take several moves/rounds between trains to be achieved.

The work in (Lowry and Munro 2002; Lowry 2004) proposes a new mechanism to perform detection of isolated trains, i.e., trains that contain only garbage objects. The original mechanism for performing this is deemed incorrect, subject to race conditions with the distributed pointer tracking mechanism. This may result in wrongfully identifying a train as isolated when in fact it has reachable objects. The solution introduces the notions of open and closed trains. Trains oscillate between open and closed state. Cars can only be inserted in a train when it is open. This imposes additional synchronization with pointer tracking while ensuring safety.

III.1.2.3.7 Cycle Detection with Group Merger

Distributed cycle detection within groups of processes is also addressed in (Rodrigues and Jones 1998), with further detail on mechanisms for dynamic group creation and management. Groups are created to be scanned as a whole and detect cycles exclusively comprised within them. Groups of processes can also be merged and synchronized so that ongoing concurrent detections can be re-used and combined.

Cycle detection is preceded by group creation, which is initiated with candidate selection. When a candidate is selected, two strictly ordered distributed phases must be performed to trace objects: mark-red and scan phases. Mark-red phase paints the distributed transitive closure of the suspect objects with the color red. This must be performed for every cycle candidate. Termination of this phase creates a group. Afterwards, the scan-phase can be started independently in each of the participating processes.

The scan-phase ensures that suspected objects are indeed un-reachable. Objects that are also reachable from other processes (outside the group) are conservatively marked green (i.e., not garbage). Green marks must be propagated to other processes in the group, following the local and remote references. This consists of alternating local and remote steps. When two group detections meet, they can either: i) merge, ii) overlap, or iii) retreat. Retreat happens when one collection in mark-red phase meets a collection already in scan-phase (mark-green).

This algorithm has fewer synchronization requirements w.r.t. (Lang et al. 1992; Rodrigues and Jones 1996), since it does not trace all objects in the processes, but only those belonging to a potential cycle. The cycle detector must inspect objects individually. This demands strong integration and cross-dependency with the execution environment and the local garbage collector. Mutator accesses to objects involved in a group detection, during mark-green in scan phase, can raise race conditions similar to tri-color local tracing. To ensure safety, all of an object descendants may need to atomically be marked green. This blocks the application when it is actually accessing the objects. Processes need to store state about all ongoing group detections comprising them.

III.1.2.3.8 Mark Propagation With Optimistic Back-Tracing

In (Fessant 2001), marks associated both with stubs and scions are continuously propagated between sites as all cycles are detected. Instead of time-stamps as in (Hughes 1985), marks are complex holding three fields (distance, range and generator identifier) and an additional color field. Marks originate from generators that include scions and GC local-roots. Generators are considered lexicographically ordered with GC local-roots as upper bound. Marks are also totally ordered according to their generators. Marks from the same generator are considered greater if they have smaller distance field.

GC local-roots first, and then scions, are sorted according to these marks. Objects are traced twice every time the LGC runs starting from GC local-roots and scions: first in decreasing, and then in increasing order of marks, towards stubs. Stubs require two marks. Mark propagation through objects to the stubs is decided by min-max marking (this is heavier than simply reach-bit propagation). One message propagates marks from stubs to scions. Thus, the two marks associated with each stub are respectively the greatest and the smallest mark propagated to it. Marks propagate from stub to corresponding scions by messages. Stub messages need to include, besides marks, additional information about every single sub-generator reaching each stub.

Cycle detection is started by generators that propagate marks, initiating in GC local-roots and scions recently created or touched by the mutator. When a remote invocation takes place, a new generator is created and its associated mark must be propagated along the downstream distributed sub-graph. Generator records include creation time, a maximum range field (similar to a time-to-live field) and a locator of the mark generator. White marks represent pure marks, while gray marks indicate mixing of marks from different generators during a local trace.

When a generator receives back its own mark, colored white, a cycle has been safely detected. If instead the mark is gray, this means other sub-paths lead to the same scion and sub-generations must then be initiated. Sub-generators are created in the back-trace of the generator that receives the gray mark. This lazy back-tracing mechanism can be very slow. To accelerate this, an optimistic variation leverages knowledge about sub-generators triggering several back-traces in different processes. Several back-traces can be performed in parallel.

Optimistic back-tracing is more efficient, yet, unsafe without further cautions. Possible errors are prevented resorting to a special black color associated with marks in scions whose sub-generator status is later revised. This ensures safety.

The resulting global approach detects all cycles simultaneously. While it avoids need to initiate one detection each specific cycle candidate, cycle detection is achieved at the expense of additional complexity and performance penalties. The mark propagation consists of a global task being continuously performed; it has a permanent cost. Instead it could be deferred in time, and executed less frequently. It imposes a specific, longer, heavier LGC that must collaborate with the cycle detector. Scions are always sorted, twice, and LGC performs min-max marking, which is heavier than simply reach-bit propagation.

Furthermore, there is a tight connection and dependency among LGC, acyclic DGC and distributed cycle detection. This is inflexible since each of these aspects is subject to optimization in very different ways, and should not be limited by decisions about the others.

III.1.3 GC in Transactional and Persistent Systems

This section describes algorithms specifically developed for systems providing transactional semantics and persistence[4] to applications (i.e., *mutators*). These aspects are essentially orthogonal to the basic problem of identifying garbage (either directly or indirectly). Thus, it is not surprising that algorithms for these systems are, at the core, extensions of algorithms of tracing and reference-counting families. Thus, we address only the necessary adaptations specific to these systems in order to ensure safety, while preserving performance.

We present both algorithms that operate in centralized repositories, as well as those for distributed repositories (but without data replication that is addressed in the next Section). Once again, transactional semantics and persistence are orthogonal to distribution.

III.1.3.1 Atomic Garbage Collection

Fault-tolerant GC in persistent centralized, single-partition stores, was first introduced in (Kolodner et al. 1989), using copying collectors. This algorithm blocked the mutator during collection, and was made incremental in (Kolodner and Weihl 1993).

In case of failure during GC, copying-based GC algorithms expose two vulnerabilities. Forwarding pointers, copied/moved objects, or both may be lost during failure. Thus, these tasks must be performed atomically in order to be recoverable.

The GC registers in the log both the copying as well as reference-patching actions, for each scanned object. Upon recovery, the previously stable state *from-space* is recovered and the log replayed, recovering the state of *to-space*. The collector conservatively regards entries in the log of running transactions as members of the GC root-set, to prevent objects from being prematurely reclaimed before transactions commit.

An alternative approach is used in (Detlefs 1990; Detlefs 1991). Instead of logging every operation, uses virtual memory mechanisms to trap memory writes (due to forwarding pointers or mutator activity) as introduced by (Appel et al. 1988), and synchronously writes modifications to disk to ensure correctness when failures occur.

[4]In persistent systems, persistence of an object is attained through reachability from a persistent root(Blondel et al. 1998).

III.1.3.2 Fault-tolerant *Replicated* Copy

The algorithm proposed in (O'Toole et al. 1993) is based on previous work presented in (Nettles et al. 1992), enhanced with fault-tolerance. It stores original and forwarding pointers in a dedicated table, resident in memory. Thus, objects in the *from-space* are never modified by the GC, and continue to be accessed by the mutator, while reachable objects are copied to *to-space*. Thus, the GC algorithm is considered non-destructive, as opposed to atomic GC algorithms.

Modifications on objects performed by the mutator are logged. When every object has been copied, changes recorded in the log are reapplied and spaces are flipped. Fault-tolerance is thus straightforward. Upon recovery, after a failure, the *from-space* can be safely recovered from stable storage. Safety is preserved even if the table of forwarding-pointers and *to-space* are discarded. However, all the work performed by the GC during the previous iteration is lost, and must be restarted.

III.1.3.3 Transactional Reference Listing

The work in (Maheshwari 1994), was developed in the context of Thor (Liskov et al. 1992). It presents a reference-listing algorithm, following the main aspects from SSP-Chains (namely, indirect protection of objects when duplicating references, without the need to contact object owner process), while extending it to address fault-tolerance in a transactional, persistent distributed object store, with multiple object repositories. The algorithm ensures safety in the presence of transactions.

There is no data replication, only caching. An object exists at only one server and may be cached by client processes. Remote references always target objects in servers. References from objects cached at clients to objects in servers are only temporary, i.e., they exist for the duration of the (possibly distributed) transaction. Upon transaction commit, updated cached copies are sent back to the servers. Persistent references are only those among objects residing at servers.

Scions whose corresponding stubs reside at other servers are kept in stable storage and re-loaded on server recovery. Scions created on behalf of clients are not, as it would be very expensive. Instead, each server keeps a list of clients in stable storage. Upon recovery, the clients are contacted to inform of existing stubs so corresponding scions can be reconstructed.

The algorithm is fault-tolerant, and correctly manages object reachability in the context of transactions and roll-back. It requires an atomic shutdown protocol that is not easily scalable. It is used when all servers must agree when to deem a client as failed. This is needed, in order to ensure safety, when discarding scions created on behalf of the failed client. It is not complete, it does not reclaim garbage cycles comprising more than one server.

III.1.3.4 Transactional Mark-and-Sweep

Garbage collection in a centralized client-server transactional object store (Carey and DeWitt 1986) is addressed in (Amsaleg et al. 1995b; Amsaleg et al. 1999). It uses a variant of mark-and-sweep, adapted to ensure safety in transactional scenarios.

In this persistent store, clients also access objects within cached copies of database pages. Instead of transferring updated objects only when transactions commit, as in the previous algorithm, page transfer (containing objects) from client to server is allowed to happen asynchronously, in any order, before transactions are completed. This improves efficiency and reduces commit delay, since update transfer can happen in the background. These optimizations nonetheless raise safety issues w.r.t GC, namely: i) partial flush of multi-page updates, ii) transaction roll-back, iii) overwriting of collected pages.

Since the server, where the GC runs, may receive pages from clients with updated objects in any order, it may have an inconsistent representation of inter-page references (e.g., it may have a page holding an object but not the updated version of another page holding the reference to the object) and thus incorrectly reclaim reachable objects. This is specially the case with newly created objects. Moreover, in this transactional scenario, garbage is no longer a stable property during transactions, since a newly unreachable object may be resurrected if the transaction responsible aborts and rolls back.

The algorithm addresses the aforementioned problems as follows. Safety w.r.t. i) and ii) is preserved by ensuring that objects newly created, or that became unreachable, in the context of a transaction, are only eligible for collection after that transaction completes. This is implemented by conservatively including in the GC root-set, two additional tables: i) PRT-pruned reference table, that references all objects targeted by deleted references, and ii) COT-created object table, that references newly created objects. When a transaction completes, both PRT and COT are flagged for later removal. Finally, problem iii) is solved by ensuring that reclaimed space in pages is only reallocated to new objects, when the freeing of that space is already reflected in stable storage.

This algorithm is incremental, performs partitioned-tracing, and does not need to hold transactional locks, while assuming that applications access objects holding the necessary locks, in the context of ACID transactions. Fault-tolerance is handled by simply restarting GC *from scratch* on server recovery. It is not complete as it is unable to collect inter-partition garbage cycles.

III.1.3.5 PMOS: Train Algorithm for Persistent Systems

The train algorithm has also been adapted to persistent systems. In the centralized version, PMOS (Moss et al. 1996; Munro and Brown 2001), cars are pages of the database. A page is both the unit of caching and tracing. Incoming references to a car from other cars are maintained on stable storage, while outgoing references to other cars are recalculated when the page is loaded in memory, and later, also when the differences on modified pages are sent lazily back to disk.

The algorithm as its predecessor for non-persistent scenarios, operates by progressively moving referenced and referring objects, clustering them in the same car (page) or in the same train (set of pages). The algorithm is concurrent with the mutator, fault-tolerant, and complete. Disk usage is not optimized since tracing a page or train may require moving objects to different pages that may no be loaded in cache.

A combination of distributed and persistent train algorithms is presented in DP-MOS (Brodie-Tyrrell et al. 2004), for garbage collection of cluster systems.

III.1.3.6 Partitioned GC of Large Object Stores

Design and implementation of complete GC on single-site partitioned object stores is thoroughly addressed in (Maheshwari and Liskov 1997c). The algorithm was developed in the context of Thor but addresses centralized storage only. GC collects one partition at-a-time. Inter-partition references must be managed, and inter-partition cycles can occur. Inter-partition references are managed using a form of reference-listing, efficiently encoded in *trans-lists* that saves memory when storing incoming and outgoing data regarding remote references. *Trans-lists* are kept on stable storage, and changes are encoded as differences (*[sic] deltas*) for later update to disk.

Global tracing for collection of inter-partition cyclic garbage is piggy-backed on regular tracing of each partition. Each one has a mark-bit and bitmap of marked objects (markmap) that are kept on stable storage and subject to delta encoding of modifications. Objects once marked in global marking phase, are never unmarked until the phase terminates. Partition may be marked and unmarked several times, namely when it is needed to mark an object found to be targeted by a inter-partition reference not yet considered. This converges so that all partitions eventually are marked, and then marked objects can all be reclaimed as they are cyclic garbage.

The algorithm is fault-tolerant, incremental, and complete, since it detects cycles that span more than one partition. It employs specialized approaches for detection of intra-partition (tracing), acyclic inter-partition (reference-listing), and cyclic inter-partition garbage (global marking). It optimizes disk usage because it does not move objects across partitions and thus does not need to patch references as copying collectors and PMOS.

The main drawback is that (e.g., contrary to PMOS) the algorithm collects all cyclic inter-partition garbage only at the end of each global marking phase (e.g., PMOS may collect one or several cycles confined within each train). Global marking phases are thus long, retrace the same partition several times, and though incremental, will delay reclamation of all cyclic garbage.

III.1.3.7 Partition Selection Policies

Algorithms where partitions are independently traced may observe different performance, namely amount of reclaimed memory and disk activity, depending on what partition is selected to be traced, and how often tracing is performed. This has been comprehensively studied and measured in (Cook et al. 1994; Cook et al. 1998), where several heuristics for partition selection are designed and evaluated.

Significant partition selection heuristics are: i) mutated partition, ii) updated pointer, and iii) weighted pointer. Heuristic i) selects for tracing the partition where most pointers were updated since the last collection. Heuristic ii) selects the partition with the largest number of (previously) incoming references, that were deleted (overwritten) since the last collection. Heuristic

iii) refines ii) further and weights differently each deleted reference, with the depth of the targeted object. The closer it is to the root, the bigger the weight given to that reference, since it may potentially detach more objects from the reachability graph. The partition with the highest weighted sum is selected for tracing. The performance of these heuristics is compared with three other heuristics that place bounds, and assess quality of observed values: iv) random, v) oracle (simulation that always selects the partition that contains the most garbage), and vi) no-collection.

Experiments indicate that heuristics ii) and iii) are the most useful. Results show that garbage collection penalty imposed on transaction throughput is minimized using heuristic ii), being very close to perfect (oracle). Heuristic i) performs worse than no collection at all. Regarding reclaimed space, heuristic ii) is once again the one that frees more space, and sooner. This shows that a more sophisticated and complex heuristic (as iii) may not produce the best results. W.r.t. partition collection frequency, the cost of tracing too often out-weights the benefits of reclaimed garbage, that is less in each collection. Tracing too seldom, wastes space and increases disk usage by the database.

III.1.3.8 GC Consistent-Cuts for Databases

GC-consistent-cuts (Skubiszewski and Porteix 1996; Skubiszewski and Valduriez 1997) are proposed for GC of a centralized, single-partition object-oriented database (O_2). The algorithm is based on the premise that only reference overwriting (i.e., assignment with consequent modification) can create garbage and it detects these operations with resort to transactional synchronization mechanisms, such as read/share and write/exclusive locks on pages.

Armed with this information, the garbage collector constructs GC-cuts, possibly more than one in parallel. A GC-cut is a set of pages with at least one copy of every page in the database. It may hold several copies of the same page if necessary. Copies of the pages are created at different instants. While this helps to avoid mutator disruption, the various copies of the same pages may be inconsistent. They may have different content that was valid only during different transactions. The different copies of all pages in the cut, in combination with knowledge from locks ensures consistency. The algorithm does not require any barriers to monitor mutator access on pages, because it takes the necessary copies before, and after, a page is locked for writing.

If a page in the cut contains references to objects in another page, the latter must also be included in the cut. CG-cuts are thus constructed incrementally, in parallel with the mutator. When the CG-cut is complete, garbage can be identified and reclaimed. The cut is subject to mark-and-sweep. Mark and sweep phase are sequential but concurrent with mutator, since they are performed using copies of the pages. An object is considered garbage if and only if it is garbage in every page in the CG-cut where it occurs. This is because references to the object may have been modified in the context of several transactions and the algorithm cannot decide which or if any will commit. Thus, conservatively, it looks for object reachability in any of the pages. Objects found to be garbage are then deleted from the actual data pages.

The algorithm is incremental, concurrent with the mutator, and complete. However, the size of a GC-cut is potentially unbounded since it may contain more than one copy of every page in the database. Garbage reclamation is delayed until the GC-cut is complete. Thus, the amount of memory required for the GC-cut, and the floating garbage that exists until a complete cut can be constructed, may introduces memory overhead. Since it uses and strongly depends on information about database locking mechanisms, it is difficult to apply this algorithm to different environments.

III.1.4 Distributed GC in Replicated Memory Systems

This section describes garbage collection algorithms for distributed systems with data replication. Thus, several replicas of the same object may exist in different processes; the mutator in each process is only able to access locally replicated objects. There is no remote invocation of objects on other processes. These are fundamental differences w.r.t the scenario considered in the previous section.

Such replicated systems impose additional difficulties on DGC algorithms, to safely handle multiple copies of the same object and build a safe and correct view of the consolidated object graph, i.e., accounting for all the references to, and from all the replicas. This enforcement reifies a *Union Rule* (Ferreira and Shapiro 1995) that has been implemented, implicitly or explicitly, in each algorithm for replicated memory systems.

This introduces a new dimension to the GC problem as explained next. In centralized and non-replicated distributed systems, GC must worry with correctly accounting for the accesses and modifications performed by the mutator on objects, during time. This is exacerbated in distributed systems with the added asynchrony, message delay, communication failure, etc.

With replication, GC has to take into account an additional space dimension in the sense that accesses and modifications to each object are no longer being performed on a single physical entity but on a number of replicas, physically disparate, but that comprise a single logical entity. Notions such as reachability, an object being referenced by another, etc., must be extended to maintain safety of GC w.r.t. applications, despite replicated and possibly inconsistent data. These issues are also present in the context of systems that allow object caching at client processes (as some presented in the previous section), but they are less demanding since these copies only last for the duration of a transaction.

Despite their commonalities, replicated memory systems vary greatly in scale, from multiprocessors with private memory,[5] cluster-based DSM (Li and Hudak 1989)[6] architectures, to large-scale distributed replicated object stores, possibly also transactional and persistent. This, together with different knowledge, dependency, and assumptions thereof, by the GC, regarding consistency enforcement among replicas, motivate the different implementation of the *Union Rule*, by the algorithms described next. The following sections present algorithms that progressively drop assumptions about, and/or requirements imposed on the execution environment, namely consistency enforcement, geographic scale, and communication protocols.

[5]**N**on-**U**niform **M**emory **A**rchitecture.

[6]**D**istributed **S**hared **M**emory.

III.1.4.1 GC for DSM assuming memory consistency

A number of initial works assume strong object (or memory) consistency when performing garbage collection. In these systems the *Union Rule* is implicitly implemented by this assumption, as they require all memory to be consistent for GC, even w.r.t. objects that are neither being accessed nor modified by applications. This produces disruption to both GC and the mutator in applications.

The algorithm proposed by (Le Sergent and Berthomieu 1992) is a multiprocessor copying collector adapted to DSM. It does not partition memory which imposes delay on garbage reclamation, and needs to lock pages when they are being traced. The algorithm presented in (Kordale and Ahamad 1993; Kordale et al. 1993) uses a mark-and-sweep approach and also requires strong object consistency.

They interfere with the coherence engine and therefore impose severe performance penalties, both to the GC and applications. Garbage collection is delayed by requiring it to analyze only consistent data. Conversely, if the GC is allowed to lock data (especially if locks are exclusive) while examining objects, it will compete with applications, even when they are not modifying data, and slow them down.

III.1.4.2 GC for DSM non-interfering with consistency

To avoid the the intrusion and performance bottlenecks of the previous solutions, the strict requirement of memory consistency, when performing GC, must be dropped. This has been proposed initially in (Ferreira and Shapiro 1994a; Ferreira and Shapiro 1994b), and later also in (Yu and Cox 1996). These works are based on the observation that the garbage collector, in order to operate correctly, has looser consistency requirements than applications.

GC in the BMX System: The work presented in (Ferreira and Shapiro 1994a; Ferreira and Shapiro 1994b) describes the design and implementation of a replication-aware GC algorithm, for a weakly consistent, persistent distributed shared memory (DSM) system: BMX (Ferreira and Shapiro 1993). The unit of replication is a *bunch*, i.e., a pre-defined group of contiguous objects. All objects belong to a single *bunch*. A bunch may be replicated at several processes. Nonetheless, granularity of data ownership, w.r.t. DSM, is still maintained at the object level.

Each replicated *bunch* maintains its own DGC structures (stubs and scions), w.r.t: i) other bunches (inter-bunch stubs and scions), and ii) other replicas of the same bunch (intra-bunch). DGC, i.e., inter-bunch GC, uses a reference-listing algorithm, thus it is not complete. Each replicated bunch is collected by the GC, independently of other bunches, and other replicas of the same bunch.

The Union rule is enforced by *intra-bunch* stubs and scions, that bind the replicas of the same object together. These chains of stubs and scions, connecting replicas, always follow the inverse path of *owner-pointers*, used by the entry-consistency protocol (Bershad and Zekauskas 1991). An object at its owner process, even if unreachable, is never reclaimed while is still reachable in other replicas of the same bunch. As an object becomes unreachable in non-owner replicated bunches, the stubs and scions will be progressively eliminated until the object can be reclaimed. Thus, GC relies heavily on the fact that only the owner of an object can modify, or may have modified it, after the last invalidation. Armed with this knowledge, the GC is able to make progress, in spite of inconsistent data, without interfering with the mutator.

For performance reasons, the local (*intra-bunch*) GC algorithm is based on a copying collector (O'Toole et al. 1993). Thus, special care is required, within each replicated bunch, w.r.t.: i) copying replicated objects, between *from-space* and *to-space*, and ii) patching references to copied objects. Thus, a live object is only copied at its owner-node, the only one where it may be modified, thus without requiring a DSM write-token. Object scanning is conservatively performed in all (possibly inconsistent) object replicas, without requiring read-tokens. References to copied objects are patched lazily, leveraging messages that are exchanged by the consistency protocol, when a process attempts to access an object, after it has been locally invalidated.

To detect cycles of garbage objects that span more than one bunch, a partially complete approach is employed, based on group-tracing (Lang et al. 1992) on bunches. It is very limited as it is only able to detect a garbage cycle, if all its comprising bunches are replicated at the same process, which may not occur and would impose storage overhead.

GC in TreadMarks The work described in (Yu and Cox 1996) also addresses DGC in the context of a distributed shared memory system, TreadMarks (Keleher et al. 1994). It also avoids interference with consistency enforcement and does not require locking objects, while these are being traced. It leverages information exchanged between processes, by the lazy-release consistency protocol (Keleher et al. 1992). Thus, this solution is closely tied with the consistency protocol used, and is not general. The authors employ a conservative collector (adequate for weakly-typed languages as C) that uses partitioned garbage collection. In this case, a partition is always an entire process.

All messages exchanged between processes are scanned for possible contained references. The DGC algorithm used is weighted reference-counting, prone to message duplication and weight underflow. Inter-process references are managed via *import* and *export* tables. The Union Rule is enforced in a similar way as the previous algorithm, by *depart* tables in each process. Each entry in this table, refers to an object that has been previously replicated and modified, but whose changes have not been propagated to the owner node yet. These objects are regarded as roots for the LGC. Once again, this leverages knowledge about, and relies on, the consistency protocol used.

The DGC algorithm used is not complete. Furthermore, the system is unable to promptly reclaim any cycles of garbage that span processes. It must stop all the mutators and enroll in a global sequential marking scheme, which is clearly unscalable, and thus actually impracticable.

III.1.4.3 Garbage Collection in *Larchant*

The algorithms presented in the previous section aim at minimizing interference with the memory consistency protocol used. Nonetheless, they were designed while knowing, *a priori*, how consistency would be enforced. Therefore, they take advantage of specific information about protocols to decide the moments when to perform GC work, while avoiding hindering consistency enforcement and applications. Thus, while they do not interfere with consistency, they are in fact dependent of a specific consistency protocol used, because they were designed to integrate with it.

The mechanisms used are relatively inexpensive in the context of DSM systems. However, they are largely impractical in other scenarios, where processes run in different geographic locations. This stems from the fact that the consistency protocols employed are not scalable outside cluster environments (i.e., local area networks, with low latency and high bandwidth).

These assumptions must therefore be dropped in the context of large-scale systems where the consistency protocols are not as strict (e.g., optimistic (Saito and Shapiro 2005)), or if the GC algorithm is to be made completely general and independent irrespective of it. This was first recognized in the context of the Larchant project (Ferreira and Shapiro 1995; Ferreira and Shapiro 1996; Ferreira and Shapiro 1998). This work coined and formalized the *Union Rule*.

This work proposed a new DGC algorithm based on a set of safety rules that take into account the existence of replicated objects, in a well-defined manner. In particular, the Union Rule explicitly introduced, ensures the safety of the DGC algorithm, while not depending on any information regarding any consistency protocol used.

The DGC algorithm uses reference-listing (Birrell et al. 1993b). All messages carrying objects are scanned to account for references. Messages exchanged by the DGC, regarding creation and deletion of inter-process references, are batched and sent lazily. For safety though, all scion *insertion* messages must be sent before any scion *removal* messages, regarding the same object, using a FIFO channel, which is generally assumed (provided by TCP).

The Union Rule is enforced using a special message, *union* message, sent to inform an object owner of the all references contained in each of its replicas, located in other processes. The owner node thus centralizes stubs regarding references contained in all replicas of an object. Therefore, only the owner node is able to send messages regarding scion removal, when another object is no longer referenced by any of the replicas.

To avoid distributed races, during reference creation and deletion, involving replicated objects, scion-insertion and removal messages must be delivered in causal order, as the following example illustrates. When an object y is replicated in two processes (e.g., to $P1$, from owner $P2$), containing reference to another object, $z(P3)$, $P1$ is in charge of sending an *insertion* message to $P3$, that may be sent lazily. If later, both replicas of y are modified so that they no longer contain references to $z(P3)$, $P2$ will send the corresponding *removal* message, only after it has received the *union* message from $P1$, to uphold the Union Rule. However, since there are three processes interacting, the *removal* messages from $P2$ to $P3$, although being sent after the *insertion* message from $p1$ to $P3$, could arrive earlier, breaking algorithm safety. Therefore, these messages must be delivered causally.

The Larchant prototype follows previous work (Ferreira and Shapiro 1994a). It allocates objects within bunches and still uses an entry-consistency protocol. Nonetheless, the algorithm is sufficiently general to be applied to other consistency protocols. Larchant allows multiple-writer scenarios, and is oblivious of any locking information, while still encompassing the notion of object owner, i.e., the process that is allowed to propagate an object to other processes.

Nonetheless, contrary to previous work, this process may not necessarily be the one with writing permission. The causality requirements, regarding delivery of creation and deletion messages, may be cheaply implemented, in the context of DSM, where only the object owner can modify it. Cycle detection uses the same approach as in the BMX system.

Larchant has been the basis for the DGC algorithm implemented in the PerDiS project (Blondel et al. 1998; Ferreira et al. 2000). However, the solution proposed for DGC is not scalable to wide-area networks, because it requires the underlying communication layer to support causal delivery. In addition, it is not complete as it fails to detect and reclaim cycles of garbage, in general.

III.1.4.4 DGC for Wide Area Replicated Memory

The work described in (Sanchez et al. 2001) drops the causality requirements imposed on the communication protocol by Larchant. It is based on SSP-Chains (Shapiro et al. 1992a), while remaining safe in the presence of replication.

It extends the Larchant algorithm by representing object replication in processes explicitly, using specialized structures (called `InProp` and `OutProp` tables), instead of resorting only to stubs and scions. This double nature of GC structures simplifies the design and improves the algorithm in several ways, as described next.

First, it requires fewer safety rules, just three instead of the total five prescribed by Larchant. This set of rules conservatively creates the scion-stub pairs of an inter-process reference immediately before those references are exported or imported. This algorithm also achieves better scalability, namely in wide-area scenarios, because it drops the requirement for causal message delivery, by implementing the *Union Rule* differently.

The Union Rule is enforced by demanding all processes holding replicas of an object, that are unreachable locally and from scions, to send an appropriate *Unreachable* message to the process where they were replicated from. When a process has received Unreachable messages from all the other processes that have replicated an object from it, and its replica is also unreachable except from OutProp entries, it sends, lazily, *Reclaim* messages to other processes holding replicas of the object. As a result, all the InProp and OutProp entries regarding the object are cleared, and all the replicas are independently reclaimed, the next time an LGC runs in each process.

The algorithm DGC-WARM does not depend on any information about consistency for safety, and does not rely on the notion of owner process. While in Larchant, only the object owner may replicate the object to another process, this work drops that requirement and allows any process to propagate object replicas. Thus, it must prevent reclamation of all other objects referenced from any of the replicas. Additionally, there is no point where the DGC information about a specific object (i.e., stubs regarding the references enclosed in all its replicas) is centralized. Stubs are scattered by the processes holding the actual references in each replica.

This algorithm it is not complete w.r.t. distributed cycles of garbage comprising replicated objects. Since the contributions of this dissertation comprise a cycle detector for this algorithm, its underlying model will be described in greater detail in the next chapter.

III.1.5 Conclusion

Performance Issues The performance impact of local garbage collection has been thoroughly addressed in work by (Zorn 1989; Zorn 1990a; Zorn 1990b; Zorn 1993), that the reader may refer to. This continuous work implements and compares a number of alternative algorithms and other integration mechanisms (barriers, conservativeness) used, w.r.t. CPU overhead, reference locality, memory overhead.

However, there is no *standard* benchmark for testing and comparing distributed garbage collection solutions, such as there is OO7 (Carey et al. 1993) for comparing object-oriented databases, and TPC-C[7] for relational databases.

Recent work in (Blackburn et al. 2004a) confirms performance benefits of garbage collection over manual memory management because contiguous allocation out-performs free-list allocation. Present architectural trends will accentuate this advantage in the future. Garbage collection even out-performs no memory management because it improves program locality, as live objects are maintained close to each other in memory, confirming results of previous work in more recent architectures. Generational collectors perform widely better, due to reduced collection time (as opposed to whole-heap GC that degrades mark-and-sweep performance greatly). Therefore they have been adopted in the two major virtual machines (Java and .Net).

The work in (Huang et al. 2004) further improves program locality, while monitoring actual program behavior when transversing object graphs. The GC leverages this information to co-locate groups of objects whose references among them are more frequently transversed. This technique is named OOR-online object reordering.

GC-Spy (Printezis and Jones 2002) is a generic heap visualization framework initially developed for Java. It allows fine-grained inspection of the memory usage in the virtual machine and LGC behavior, supporting: i) visualization of different heaps independently, ii) inspecting trains, card-tables, free-lists, and iii) monitoring different GC-generations (e.g., nursery, mature space). It has been recently ported to Rotor (Marion and Jones 2005).

The Memory Management Toolkit (Blackburn et al. 2004b) is a portable and extensible framework that allows the development and evaluation of algorithms for LGC. It extends the Java type system in order to implement memory addresses (instead of opaque references) in Java. The MMTk provides a fair quantity of built-in GC *mechanisms* (e.g., free-list allocation, large-object allocation) and *policies* (e.g., mark-sweep collection, copying collection, reference-counting collection), which enable the design of new LGC algorithms by composition of policies (e.g., Ulterior Reference Counting described in Section III.1.1.3.4).

The evaluation of GC in persistent stores has been addressed in (Amsaleg et al. 1995a), pointing out there is no benchmark established, but for centralized systems (Carey et al. 1993). Accordingly, the evaluation of distributed garbage collection algorithms has not been subjected to the same exhaustive study as local GC. There is no widely used benchmark for DGC, and the algorithms in the literature have been evaluated against others w.r.t. completeness, asynchrony, scalability, messages exchanged, and disruption to mutator and local GC (Plainfossé and Shapiro

[7]Transactions-per-minute, from the Transaction Processing Council.

1995; Jones and Lins 1996; Abdullahi and Ringwood 1998; Shapiro et al. 2000; Ferreira and Veiga 2005).

Unification of Tracing And Reference-Counting: Recent work has proposed that the two traditional distinct families of GC algorithms (tracing and reference-counting) are, in fact, duals of one another (Bacon et al. 2004). Both can be analyzed from a tracing, as well as a reference-counting perspective. Tracing operates on live objects (i.e. *matter*), while reference-counting operates on dead objects (*anti-matter*).

In tracing, transversal starts with mutator roots and detects all live objects. Tracing can be regarded as a one-bit sticky reference count, in which it initializes reference-counts of objects to zero and increments (only once) them when they are referenced from live objects. It requires and extra sweep phase to collect dead objects

In reference-counting, transversal starts with anti-roots (that includes all objects whose reference-count has been decremented to zero), and detects (i.e., in some sense, traces) further downstream objects whose reference-count will reach zero. This is the complemental graph of live objects except for cyclic garbage. In each iteration, reference-counts of objects are greater than or equal to the *real* value. Therefore, GC decrements them when they are referenced from garbage objects. It needs an extra-phase as well, in this case, to detect cycles (e.g., trial deletion).

Most high-performance GC algorithms are hybrids of some kind even when it this hybridization is not trivial. In Deferred Reference Counting, Zero Count Tables (ZCT) are in fact tracing elements. In Generational GC, *remembered-sets* are reference-counting elements; they are a set-representation of non-zero reference-counts, complementary to a ZCT.

Distributed Garbage Collection for Active Objects: Distributed Garbage Collection has also been applied in the contexts of *active objects*, *actors*, and agents. These concepts provide a unit of encapsulation for both data (state) and task behavior (thread of control flow). They have been employed for cooperative multitasking, namely in the field of Grid Computing.

In this context, the role of the DGC algorithm is to identify, terminate, and reclaim memory and resources owned by inactive objects, i.e., those that are unreachable via root objects, and blocked, without the possibility of further communication with other *active objects* or with users. Fundamental and recent work on DGC for *active objects* includes (Hudak and Keller 1982; Kafura et al. 1990; Kafura et al. 1995; Vardhan and Agha 2002; Wang and Varela 2006).

Final Remarks: Equivalence of distributed termination and distributed garbage collection algorithms has been demonstrated in (Tel et al. 1987; Tel and Mattern 1993; Norcross et al. 2005). This approach is also used in (Lowry and Munro 2002) to derive the mechanism for identification of isolated trains.

Recently, in (Moreau et al. 2003; Moreau et al. 2005) a complete proof of safety and liveness is provided for *Birrel's* version of the reference listing algorithm (Birrell et al. 1993a; Birrell et al. 1993b) on which Java RMI DGC is based. Although this algorithm is not complete, as it cannot detect distributed cycles of garbage objects, it is arguably the most widely deployed DGC algorithm. Remoting services (McLean et al. 2002) in .Net uses an unsafe lease-based approach to DGC.

Garbage collection is a cross-cutting issue in systems research. LGC deals with programming correctness and system performance. DGC research motivates and exercises key aspects of distributed algorithm design, such as: safety, asynchrony, scalability, liveness, termination, and completeness. A three-tiered approach to garbage collection (local, acyclic and cyclic DGC) further leverages these aspects. While many solutions have been presented along the years, on-going research demonstrates that there still remain open issues on how to balance desired properties for GC algorithms. Thus, GC is interesting in a theoretical as well as practical perspective.

Summary of Chapter: In this chapter, we presented a thorough survey of automatic memory management algorithms and/or techniques, known as garbage collection (GC). Initially, we briefly overview GC for centralized systems, local garbage collection (LGC), and its main algorithmic families that were defined in early times: i) reference-counting, ii) tracing and its variations (mark-and-sweep, and copying collectors), and iii) hybrid approaches of both (e.g., train, and generational GC).

Then, we comprehensively addressed distributed garbage collection algorithms (DGC), applicable to distributed object systems based on remote invocation, to manage distributed graphs of objects. We presented algorithms based on adaptations, to the distributed scenario, of reference-counting (e.g., distributed reference-counting, weighted reference-counting, reference-listing), and tracing (e.g., mark-sweep using time-stamps, logically centralized tracing), and a fair number of influential specialized approaches to detection of distributed cycles of garbage (e.g., group-based detection, back-tracing, group-merger) which is a hard problem in distributed systems, w.r.t. ensuring both completeness and scalability.

We also covered the most relevant approaches to GC in persistent and transactional systems, and the new problems they raise, respectively, garbage continues to exist even after applications terminate, and avoiding premature reclamation of objects that may become reachable again due to transaction roll-back.

Finally, we addressed DGC in replicated memory systems which is a relatively more recent field in GC, raising issues regarding safety and interference with consistency mechanisms. The safety of DGC in presence of replicated, possibly divergent data, is handled by the use of the Union Rule. Dealing with consistency mechanisms is accomplished by either leveraging knowledge of their operation in order to minimize interference between GC and applications, or by aiming to be completely orthogonal to them. This last option has been employed in the context of wide area networks (e.g., PerDiS project).

We concluded with some considerations regarding performance issues and recent work focusing on the unification of the two original GC families: reference-counting and tracing.

III.2 Algorithms

This chapter describes three novel approaches for distributed garbage collection. They address the issue of completeness, i.e., how to identify and reclaim distributed cycles of garbage. They are combined approaches, in the sense that each of them comprises a detector of distributed cycles of garbage, while assuming a pre-existing acyclic DGC algorithm (e.g., reference-listing, and DGC-WARM).

The first two approaches are distributed cycle detectors that target distributed systems without replication: i) one centralized approach named DGC-Consistent Cuts (Veiga and Ferreira 2003a), and ii) one de-centralized approach named Algebra-based Distributed Cycle Detection (Veiga and Ferreira 2005a). The third one comprises the first viable solution to complete distributed garbage collection for replicated object systems, DGC-Cuts for Replicated Objects (Veiga and Ferreira 2005b).

Once a distributed garbage cycle is detected, the cycle detector instructs the acyclic DGC component to explicitly delete some of its data structures (e.g., scions) that are preventing the cycle from being collected. This technique is employed by all three algorithms presented in this chapter. Once these structures are deleted, the cycle is broken into acyclic garbage. During the following iterations of the acyclic DGC algorithm, the remaining of the cycle will be collected. This operation of explicit deletion of DGC structures is safe, provided that the distributed cycle detector is sound. This stems from the observation that: if the objects belonging to the distributed cycle are indeed garbage, they are already unreachable to the mutator. They simply have not been identified as such, yet. Thus, program correctness is not affected.

The three algorithms have common goals. Safety and completeness are natural ones. Additional ones include scalability, and non-interference with the operation of both mutators and other GC components, namely, the imposition of specific solutions to local GC or to acyclic DGC. Therefore, the proposed cycle detection algorithms must be general, in the sense that they can be applicable to other existing systems.

Detection of distributed garbage cycles should be performed with lower priority, since cyclic distributed garbage is less frequent and created at a lower rate. Therefore, cycle detection, while required to be present for completeness, should not delay the more common cases in program execution (mutator, LGC, and acyclic DGC) that are executed more often, and should not be hindered or slowed down to account for cycle detection. In essence, they should be kept oblivious of cycle detection existence.

The algorithms share some data structures (e.g., stubs, scions) and constructs (e.g., graph summarization), while having adaptations to each case. Thus, for the sake of brevity, they are not thoroughly explained in all sections. Instead, the extensions and/or modifications required are highlighted.

In the remaining of this chapter, we introduce, in sequence, each of the algorithms proposed, in a dedicated section. Each of them starts with a brief global overview of the main ideas behind the algorithm, followed by the presentation of the algorithm (its data structures, cycle detection technique), and a prototypical example of its functioning. In Section III.2.3, cycle detection is preceded by an overview of the acyclic DGC for replicated object systems it integrates with, which is an extension of DGC-WARM (Sanchez et al. 2001). Each section ends with an analysis of algorithm properties. Other aspects regarding actual algorithm implementation, and runtime support for its deployment, are addressed in the next chapter.

III.2.1 DGC-Consistent Cuts

This section presents DGC-Consistent Cuts (Veiga and Ferreira 2003a; Pereira et al. 2006). This approach is employed to perform detection of distributed cycles of garbage spanning groups of processes. It introduces the notion of DGC-consistent cut, that may be regarded as the extension of the GC-consistent-cut notion to a distributed scenario. GC-consistent cuts were initially developed for centralized object databases (Skubiszewski and Valduriez 1997).

The algorithm detects distributed cyclic garbage occurring within a DGC-consistent cut. It achieves this by employing a centralized approach, in the sense that there is a designated server, the distributed cycles detector (**DCD**), which is contacted by application processes. It is responsible for constructing DGC-consistent cuts, and detecting garbage cycles enclosed in them. The detection of distributed cycles of garbage works on a view of the global distributed graph (i.e., the DGC-consistent cut) that is consistent for GC purposes. Furthermore, it is less restrictive than a consistent cut, as defined by Lamport, w.r.t. causality among application processes.

Conceptually, the algorithm constructs a DGC-consistent cut by combining representations of object graphs received from application processes. Then, the algorithm performs a conservative mark-and-sweep on the DGC-consistent cut. DGC-consistent cuts are optimized by means of a technique referred as graph summarization. DGC-consistent cuts can be obtained without requiring any distributed synchronization among the processes involved.

The algorithm eliminates limitations found in previous centralized approaches (Liskov and Ladin 1986; Ladin and Liskov 1992), by neither requiring global clock synchronization among participating processes, nor imposing bounded message latency. It also has fewer requirements w.r.t. synchronization, and maintenance of state regarding cycle detections, when compared with other group-based approaches (Lang et al. 1992; Rodrigues and Jones 1998). More details are presented in Chapter III.4, regarding the evaluation of the algorithm, and comparison with related work.

III.2.1.1 Algorithm

Figure III.2.1 presents an overview of distributed cycle detection in an example situation, which comprises four processes running a distributed application. We assume there is a pre-existing acyclic DGC algorithm deployed (e.g., reference-listing) and, thus, each process already has a DGC component running.

Each process stores its DGC structures and, periodically, sends information about them to other processes (e.g., *NewSetsStubs* messages). Using these messages, processes cooperate, by pairwise interaction, to detect acyclic distributed garbage. This pairwise interaction needs not be two-sided, i.e., a process $P1$ that sends DGC messages to another process $P2$, may or may not receive DGC messages from it.

Since this mechanism based on reference-listing is not complete, there is a distributed cycles detector (DCD), for cyclic DGC. Each process seldom sends to the DCD a conceptual representation of its enclosed object graph. This takes the shape of a compressed snapshot of the process. This snapshot also includes information about DGC structures, extended in a way described in

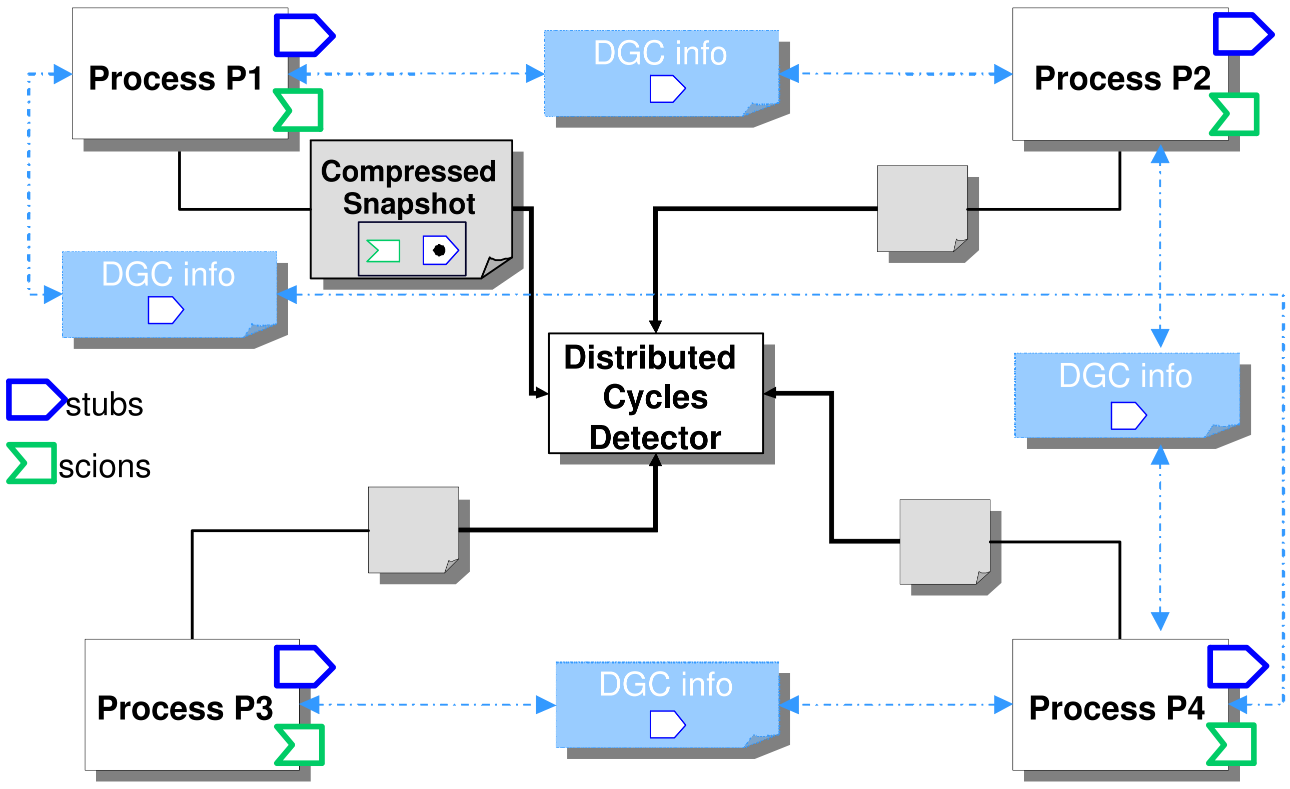


Figure III.2.1: Integration of the distributed garbage cycles detector with the reference-listing algorithm.

the next section. Armed with this information, the DCD performs the construction of a DGC-consistent cut, upon which it executes a conservative mark-and-sweep (**CMS**). After the CMS, the DCD is able to identify DGC structures (i.e., scions) keeping cyclic garbage from being collected and instruct their deletion.

After the DCD receives a compressed snapshot from a process yet unknown to it (thus widening the scope of detection), or an updated version of a compressed snapshot it already holds, it can construct a new DGC-consistent cut and perform CMS on it. The DCD discards older versions of compressed snapshots from each process, no longer in use, when it receives a more recent one. If it receives a message carrying a snapshot from a process that is older than what it already holds, it simply ignores it. Thus, incorporating snapshots always improves the recency and scope of the DGC-consistent cut, and allows the algorithm to progress, thus possibly detecting more distributed cycles.

A DGC-consistent cut necessarily portrays a situation that occurred somewhere in the past, either recent or remote. The recency of a DGC-consistent cut, nonetheless, does not raise safety issues because garbage is stable.

III.2.1.1.1 Data Structures

The DCD uses data structures already present in the acyclic DGC algorithm and defines new ones that are exclusive to cycle detection. The data structures managed by the algorithm are:

- **Scion**: represents an incoming inter-process reference. In addition to the fields already described in Chapter I.2, it is extended with a **time-stamp**. This is a numeric value provided by a monotonic counter global to the enclosing process, when the scion is created.
- **Stub**: represents an out-going inter-process reference. It is also extended with a **time-stamp** field that is equal to the time-stamp of its corresponding scion.
- **Vector-clock**: each process maintains a record of the highest time-stamp associated to scions from other processes it knows of, including itself, thus constituting a vector-clock (Mattern 1989).
- **Snapshot**: representation of the object graph of a process, including its stubs, scions, and the vector-clock maintained by the process.
- **DGC-Consistent Cut**: a conservative juxtaposition of snapshots taken at uncoordinated times, comprised of, at most, one snapshot concerning each process.

Scions, stubs, vector-clocks and snapshots are created and maintained at application processes. DGC-consistent cuts exist exclusively in the context of the DCD.

The extension of stubs and scions just described is necessary for the purpose of DGC-consistent cut creation. Nevertheless, it is also present in previous approaches to reference-listing (Hughes 1985; Shapiro et al. 1990), in *NewSetStubs* messages, to avoid race conditions, and can thus be leveraged.

NewSetStubs messages are time-stamped, with the highest scion time-stamp that the sender process was aware of, when the set of stubs was generated. This associates a view of outgoing inter-process references with the time it was taken to ensure its consistency. It ensures safety, preventing the receiving process from incorrectly eliminating scions, whose corresponding stubs were not yet created when the new set of stubs was generated (e.g., a remote invocation whose reply message was still in transit).

III.2.1.1.2 Messages

The algorithm defines two types of messages: one that is informative and another one that is operative:

- **NewSnapshot**: message sent from the cyclic DGC component of an application process, to the DCD, carrying a snapshot.[1] It can be sent lazily after a new, updated snapshot, becomes available in the process.
- **DeleteScion**: message sent from a DCD to the cyclic DGC component of an application process, instructing it to delete a specific scion that is found to belong to cyclic garbage.

[1]Actually, there may be more than one DCD process running. This and other optimizations are described in Section III.2.1.1.4.

Upon receiving a NewSnapshot message from a process, the DCD is able to infer, from the sender identifier and the enclosed vector clock, whether there was a snapshot from that process previously available. If not, it determines if the one received is indeed more recent then the previous one. This verification handles possible message reordering correctly.

Upon receiving a DeleteScion message, an application process removes the scion indicated. Once a distributed cycle of garbage is broken this way, the process may still receive NewSetStubs messages from the process where the corresponding stub resides, still containing it. This is because it may take several iterations of acyclic DGC to reclaim the rest of the broken cycle. Nonetheless, this does not hinder correctness. Since the scion no longer exists, it will not be recreated (as no more references to that object will be created because it is garbage), and the stub enclosed in the message is simply ignored (as it is would normally be by the reference-listing DGC component).

III.2.1.1.3 Cycle Detection

Cycle detection is performed by the DCD. It is divided in four distinct phases: i) snapshot reception, ii) DGC-consistent cut creation, iii) conservative mark-and-sweep (CMS), and iv) sending of DeleteScion messages.

Snapshot Reception: The DCD is always ready to receive snapshots from processes. When a snapshot is received, it is stored as a new version concerning the sender process. Therefore, the DCD may perform Snapshot Reception concurrently with another phase.

Thus, reception of a newer snapshot from a process never causes any of the other phases to stop, if they are already in progress. Naturally, when the DCD is otherwise idle, the reception of a new snapshot may trigger phase ii). When a snapshot from a process is no longer neither involved in phases ii) nor iii), it can be deleted by DCD if there is a newer version available.

DGC-Consistent Cut Creation: This is the crucial and most novel aspect of the algorithm. A DGC-consistent cut is created by the DCD by assembling a set of snapshots received from application processes. Note than a DGC-consistent cut needs not contain a snapshot from all processes. This may affect the scope of detection but not its correctness. Cycles comprising objects in such missing processes are not represented entirely. However, all other cycles are.

The difficulty in creating DGC-consistent cuts stems from the fact that snapshots from some application processes may not be available, and that snapshots actually received from application processes may have been created at uncoordinated times.

Smaller cuts comprising groups of processes that update their snapshots more often may be created to increase detection promptness. Larger cuts spanning all processes that send snapshots to the DCD will ensure completeness, but need not be created frequently.

A DGC-consistent cut is created without requiring a distributed consensus (Fisher et al. 1985) among the applications processes that send their graph descriptions to the DCD. It can be used for cycle detection, even if its global view of the distributed object graph is made of local

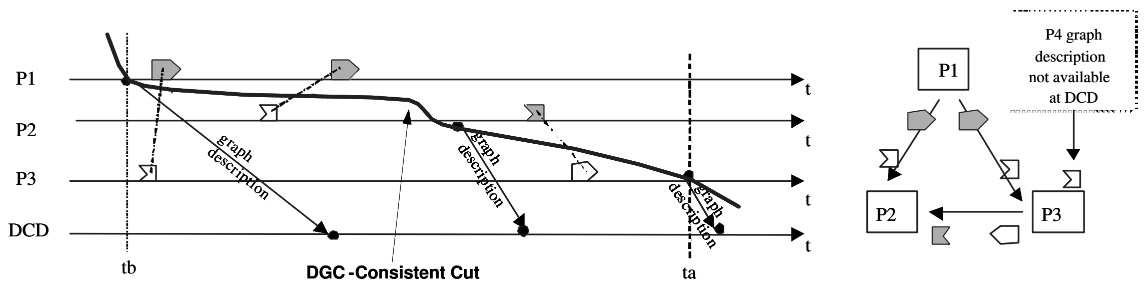


Figure III.2.2: DGC-Consistent Cut and process snapshots as seen by the DCD.

graph descriptions (sent by the application processes) gathered at different and uncoordinated moments. A DGC-consistent cut, while not required to be causally consistent, is still consistent for GC purposes.

When a snapshot is included in a new DGC-consistent cut it is not copied but just referenced, since the DCD never modifies them. It can thus be shared by several cuts. Once a snapshot is included, newer versions of the snapshot will not be considered by ongoing creations. This ensures liveness even in the improbable situation of a continuous flow of snapshot reception from several processes.

Once a DGC-consistent cut is created, it can be forwarded to the next stage, i.e., phase iii), or stored to be later combined with other DGC-consistent cuts (more details in the remaining of this section).

In Figure III.2.2, on the left-hand side, we show in bold a cut that is not consistent for typical DGC purposes; it results from the uncoordinated creation and sending of snapshots from each application process to the DCD. It is clear that this cut is such that the creation of stubs and scions is not consistent for DGC purposes; in addition, this cut does not correspond to a Lamport's consistent cut. Object graphs received by the DCD provide a view of the global graph that does not correspond to a real one; the differences are due to the shaded stubs and scions. LGCs are not represented as they can occur at any time.

However, based on such graphs, the DCD builds a DGC-consistent cut that allows it to detect distributed cycles of garbage safely. This cut is consistent w.r.t. the finding of such cycles. The line in bold represents the DGC-consistent cut. On the right-hand side of Figure III.2.2, the global graph as perceived by the DCD, based on the graph descriptions received thus far, is represented; shaded stubs and scions do not exist.

Thus, a DGC-consistent cut is a group of scions and stubs that provide a safe view of the distributed object graph. This group of stubs and scions provide a safe view of the distributed graph as long as the rules to define the root-set of the CMS (performed by the DCD) are respected. In particular, these rules specify which scions are members of the root-set of the CMS, as explained next.

Conservative Mark-and-Sweep: This phase is performed after a new DGC-consistent cut has been created. Initially, the DCD analyzes the DGC-consistent cut with special care for consistency and causality. Conservative mark and sweep is performed exclusively within the DCD, on the DGC-consistent cut. There is no exchange of messages with other processes, during this phase. The marking is performed on a separate bit-map that includes one bit per each entry in the DGC-consistent cut. This way, snapshots are untouched and may be referenced by multiple cuts.

To be safe, CMS must take into account possible, and highly probable discrepancies, among the moments when the snapshots of the different processes are taken and included in the DGC-consistent cut. There is no coordination among them w.r.t. to this activity, as they take snapshots at independent times. Thus, the CMS uses an extended set of global roots that, conservatively, includes:

1. Objects that, in each application process, are directly reachable from the GC local-roots (stack, etc.) must be obviously considered roots of the CMS.

2. Scions whose corresponding stubs are included in processes whose snapshot is not included in the DGC-consistent cut (it may even be unavailable to the DCD). These scions are members of the CMS root for safety reasons. As a matter of fact, such scions may no longer have a corresponding stub, and be removed after reception of subsequent NewSetStubs messages. Nonetheless, the DCD cannot know that for sure, using therefore a conservative approach.

3. Scions with time-stamp greater than the highest time-stamp (regarding the process where the targeted object resides), known by the process holding the corresponding stub. This is also a conservative approach. These scions are those whose corresponding stubs have not yet been created when the referring process recorded its snapshot. These scions verify the following condition:

   $scion.timestamp > VC_{P_{stub}}[P_{scion}]$

   where $scion.timestamp$ is the time-stamp given to the scion when it was created, $VC_{P_{stub}}$ is the vector-clock maintained by the process holding the corresponding stub, and P_{scion} is the identifier of the process holding the scion.

Thus, CMS is performed starting with the extended root-set, and tracing inter-process (from stubs to scions) and intra-process (from scions to stubs) references. Intra-process references are expectably more frequent and are subject to optimization. Naturally, tracing an inter-process across snapshots from two processes, can only be performed if the corresponding stub-scion pair exists in both of them, in the DGC-consistent cut. As the end result of the CMS, scions and stubs belonging to garbage are not marked.

The DCD determines whether they belong to a distributed cycle and, for each cycle detected, one of the comprised scions is selected and marked for explicit deletion. Only scions that are simultaneously garbage and, still, referenced by stubs, can belong to a distributed cycle of garbage. Then, one, any, or all of them, can be selected for deletion.

Those distributed garbage cycles that already existed when the earliest graph description (included in the DGC-consistent cut being processed) was created, and are totally included in the graph descriptions available at the DCD, are effectively detected and reclaimed. Thus, considering Figure III.2.2, all cycles that existed before `tb`, that are totally enclosed in processes P1, P2, and P3, are detected by the DCD.

Sending of DeleteScion Messages: Once a scion has been selected for explicit deletion, a DeleteScion message will be sent to the enclosing process. This message can be sent at any time, without any race condition, because the scion regards a reference that can no longer be transversed by the application, due to garbage being a stable property, i.e., once garbage, an object can never become live again. Only one message is needed to break each cycle, as the acyclic collector will then be able to reclaim all remaining objects belonging to the cycle. Messages regarding the same process may be queued and sent in batch.

During the reclamation of the remaining acyclic garbage, after a scion has been deleted, it may happen that its corresponding stub is still included in ulterior NewSetStubs messages. According to reference-listing, this does not cause the re-creation of the scion (as this can only happen via exporting a reference). Thus, the stub is ignored and will eventually disappear as the remaining acyclic garbage is reclaimed.

W.r.t each distributed cycle found, the number of DeleteScion messages sent only influences the bandwidth used and the speed of cycle reclamation. DeleteScion messages that were sent and acknowledged, are recorded, using a best-effort approach, to prevent issuing multiple messages regarding the same scion. This may happen when the same cycle is detected in more than one DGC-consistent cut.

III.2.1.1.4 Optimizations

The algorithm presented thus far is subject to three optimizations that are described in this section: i) snapshot compression, ii) use of multiple DCDs in parallel, and iii) hierarchical composition of DCDs. These optimizations do not affect algorithm safety and improve both its performance and scalability.

Snapshot Compression: Snapshots containing object graphs of processes, along with DGC information, may be very large, possibly amounting to tenths or hundreds of MB. Sending these snapshots to the DCD process would consume network bandwidth heavily, and crumble application performance and communication with other application processes (i.e., what distributed computing is all about). Furthermore, once received at the DCD process, the cumulative size of all the snapshots received would occupy a large amount of memory and disk process. Since the DCD would perform the CMS over these large snapshots, cycle detection would become a CPU-intensive operation, slowing it down drastically.

These problems are solved by summarizing the object graph (a snapshot) of each application process, prior to sending it to the DCD, in such a way that, from the point of view of the DCD, there is no loss of relevant information. This summarization compresses a snapshot of an application graph into a set of scions and stubs, with their corresponding reachability associations. As a matter of fact, neither references among objects that are strictly internal to a process,

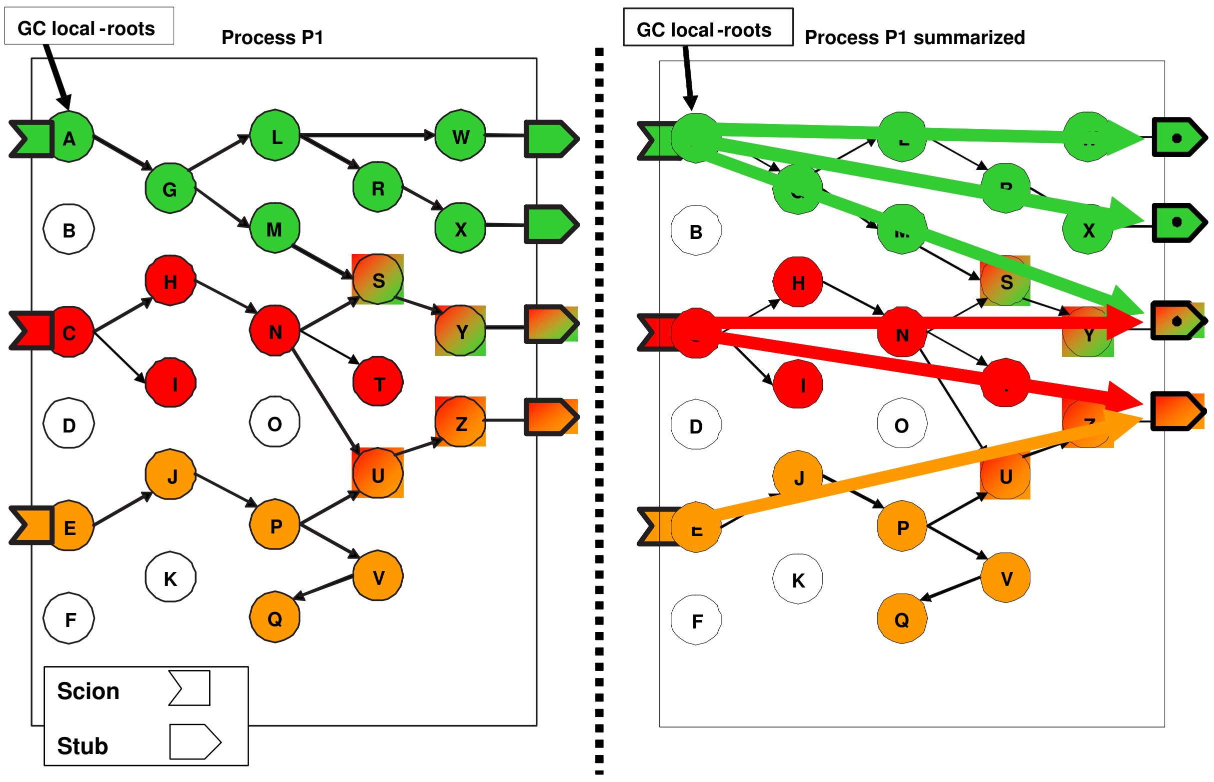


Figure III.2.3: Summarization of an object graph for snapshot compression.

nor object scalar contents, are relevant for the DCD; thus, they are not explicitly represented in the compressed snapshot. In the context of a compressed snapshot, a process may be regarded simply as a set of scions, with each one, possibly referencing one or more stubs. Additionally, each stub is marked if it is reachable from the GC local-roots. This is exemplified in Figure III.2.3.

The previous paragraph contains a simplification that will be used in the remaining of the chapter. For clarity, when we say stubs that are reachable from a scion, or from the GC local-roots, we actually mean: *stubs accounting for out-going references enclosed in objects, that are reachable from a specific object, targeted by an incoming remote reference (this one, represented by a scion), or by a GC local-root.*

This summarization is performed on every snapshot, locally to the process. The compressed snapshot is then ready to be sent to the DCD process. Thus, while processes can take snapshots by serializing local graphs, these uncompressed versions never leave the process. The DCD only uses them in their compressed form, i.e., after graph summarization.

Snapshot compression reduces bandwidth usage severely as no actual object content is transferred. The cost of transferring a snapshot is then proportional to the number of scions and stubs in a process, and independent of the actual number and/or cumulative size of the objects located in a process. Furthermore, it also minimizes complexity and memory usage at the DCD process. This allows the DCD to detect distributed cycles faster, and manage more snapshots, as opposed to what it would be able to, without snapshot compression. This allows the detection of larger distributed cycles, spanning larger numbers of processes, and comprising larger numbers of objects. Once compressed, a snapshot may be sent to several DCDs.

Multiple DCDs: Distributed cycle detection, for the purpose of cyclic garbage collection, is performed by the DCD, in a centralized manner. Notwithstanding, nothing prevents there being multiple DCD processes running at the same time, on different machines. A DCD may also be co-located in the same machine where other application processes are running, though it runs within its exclusive address space. It needs not be a dedicated machine. These issues do not affect the correctness of the solution described. They are only relevant w.r.t performance, and scalability.

Hierarchical DCDs: The DCD algorithm achieves completeness and scalability w.r.t. size of distributed cycles, using an hierarchical approach. Without it, a single DCD is clearly not able to manage snapshots from an unbounded number of processes. This, at least at a theoretical level, would limit DCD completeness, e.g., a distributed cycle spanning all processes would not be detectable, because the comprising snapshots would not fit in a single DCD process for CMS.

To circumvent this limitation, every DCD process, besides detecting distributed cycles, is also able to produce a combined, higher-level snapshot (from a DGC-consistent cut), conservatively compressing all snapshots into one that, for DGC terms, represents all processes as one. Every inter-process reference (stubs and scions) involving processes whose snapshots are not available to the DCD is maintained in the combined snapshot. However, inter-process references comprised in the snapshots included in the DGC-consistent cut can be regarded as internal references of the combined snapshot, and can therefore be summarized, as if they belonged to a single process. This is different from the group-based approach described in (Lang et al. 1992) that requires a single top-level group, containing all existing processes, tracing all reachable objects individually.

Hierarchical snapshots bound the growth of snapshot size, at the cost of only being able to detect those higher-level distributed cycles that span enclosed snapshots. Thus, distributed cycles fully comprised within a combined snapshot are not detectable at a higher-level DCD. Nonetheless, they are detectable at the lower-level DCD, where the combined snapshot was first created from an existing DGC-consistent cut.

III.2.1.2 Prototypical Example

In this section we further describe the algorithm operation with resort to three prototypical examples. In the first two, all the information required for cycle detection is available to the DCD. W.r.t. the third one, this is not the case. Therefore, this example portrays how the algorithm operates when information from some process, is either absent or outdated.

Objects are represented by their name (a letter) and their enclosing process (e.g., A_{P1}). Data structures have the process where they reside mentioned last (e.g., $Stub(F_{P2})_{P1}$, for the stub in $P1$).

Sub-graphs of connected objects may be represented in abbreviation (e.g., $\{\{A,\ C,\ B\}_{P1}, \{F,\ G,\ H\}_{P2}\}$), aggregated by its/their enclosing process. References may be also explicitly described when relevant (e.g., $B_{P1} \rightarrow\ F_{P2}$).

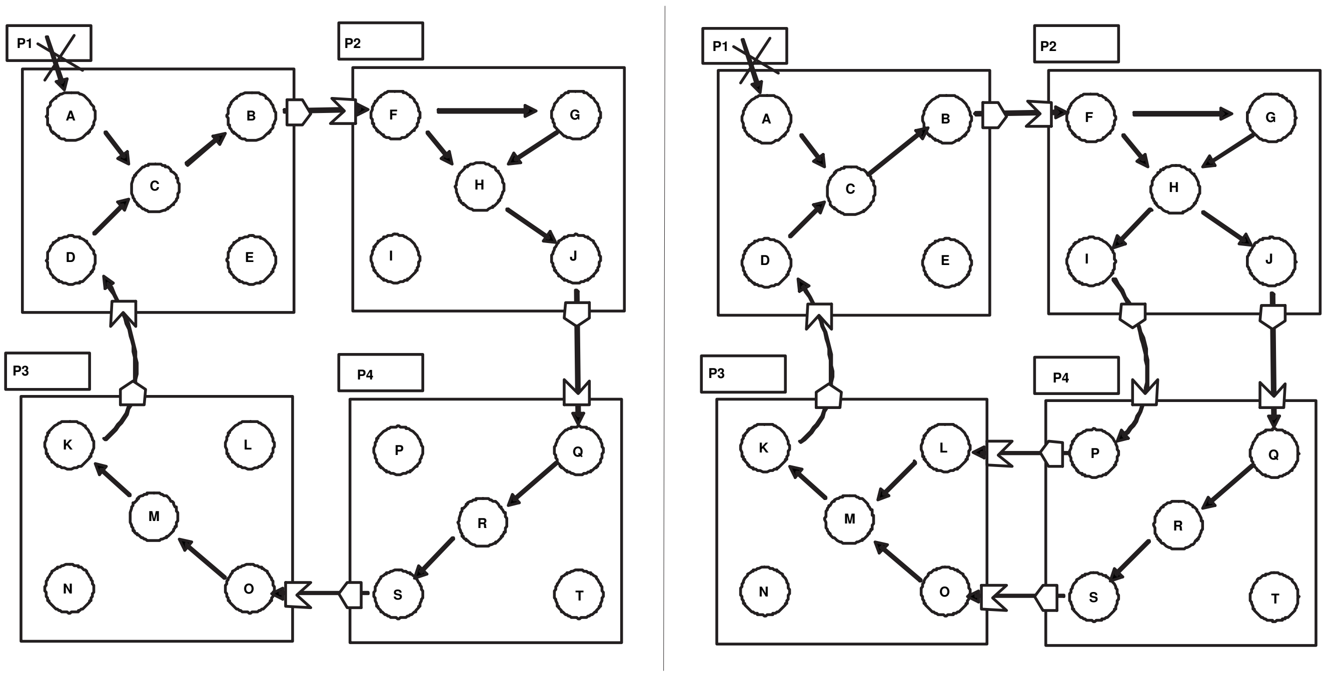


Figure III.2.4: Examples A1 and A2: distributed cycles of garbage for which the DCD has all the information available.

Example A1 and A2: In the example A1 depicted in the left-side of Figure III.2.4, we assume the DCD has already received snapshots from all processes (i.e., $P1$ .. $P4$), and constructed a DGC-consistent cut that includes all of them.

The snapshot received by the DCD from process $P1$ includes the following information (symbol $\Rightarrow$ means *evaluates to or returns*, $\equiv$ relates a field name and its value), where time-stamps are omitted for simplicity:

- $Scion(D_{P1})_{P1} \Rightarrow \{StubsFrom \equiv \{F_{P2}\}\}$
- $Stub(F_{P2})_{P1} \Rightarrow \{Local.Reach \equiv false\}$

Thus, the DCD knows there is a scion in $P1$ that leads to a stub regarding an object in $P2$. The GC local-root referencing object A_{P1} existed earlier but has has been removed by the mutator before $P1$ took its snapshot. Therefore, $Stub(F_{P2})_{P1}$ is not reachable locally. Together with the information provided by the snapshots of the other processes ($P2$..$P4$), the DCD is able to reconstruct the view depicted in the figure.

After performing the CMS on the DGC-consistent cut, the DCD has an empty root-set, since there are no stubs reachable locally. Therefore, the cycle is detected and the DCD may issue DeleteScion messages regarding any of the scions (e.g., $Scion(D_{P1})_{P1}$). From this moment on, the acyclic DGC in P1 will no longer include $Stub(F_{P2})_{P1}$ in future NewSetStubs messages. This will result in the removal of $Scion(F_{P2})_{P2}$. $P4$ and $P3$, in sequence, will eventually reclaim all objects belonging to the initial cycle.

The example A2, depicted on the right side of Figure III.2.4, portrays a similar situation, but

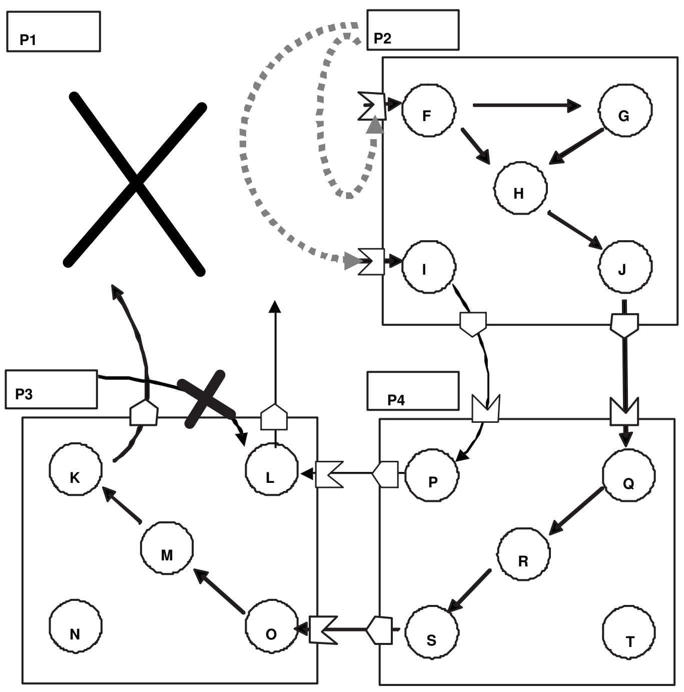


Figure III.2.5: Example B. Initial Situation: No snapshot has been received from process $P1$.

one in which there two alternate paths in the cycle. For instance, the information contained in the snapshot sent by $P4$ would be:

- $Scion(P_{P4})_{P4} \Rightarrow \{StubsFrom \equiv \{L_{P3}\}\}$
- $Scion(Q_{P4})_{P4} \Rightarrow \{StubsFrom \equiv \{O_{P3}\}\}$
- $Stub(L_{P3})_{P4} \Rightarrow \{Local.Reach \equiv false\}$
- $Stub(O_{P3})_{P4} \Rightarrow \{Local.Reach \equiv false\}$

While the information present in the snapshot from process $P3$, is:

- $Scion(L_{P3})_{P3} \Rightarrow \{StubsFrom \equiv \{D_{P1}\}\}$
- $Scion(O_{P3})_{P3} \Rightarrow \{StubsFrom \equiv \{D_{P1}\}\}$
- $Stub(D_{P1})_{P3} \Rightarrow \{Local.Reach \equiv false\}$

Even though the cyclic garbage in example A2 is more complex than that of example A1, a DGC-consistent cut comprising snapshots from the four processes is still sufficient to detect the distributed cycle. Since there no roots of reachability, because the root-set is empty, every scion in the example belongs to cyclic garbage.

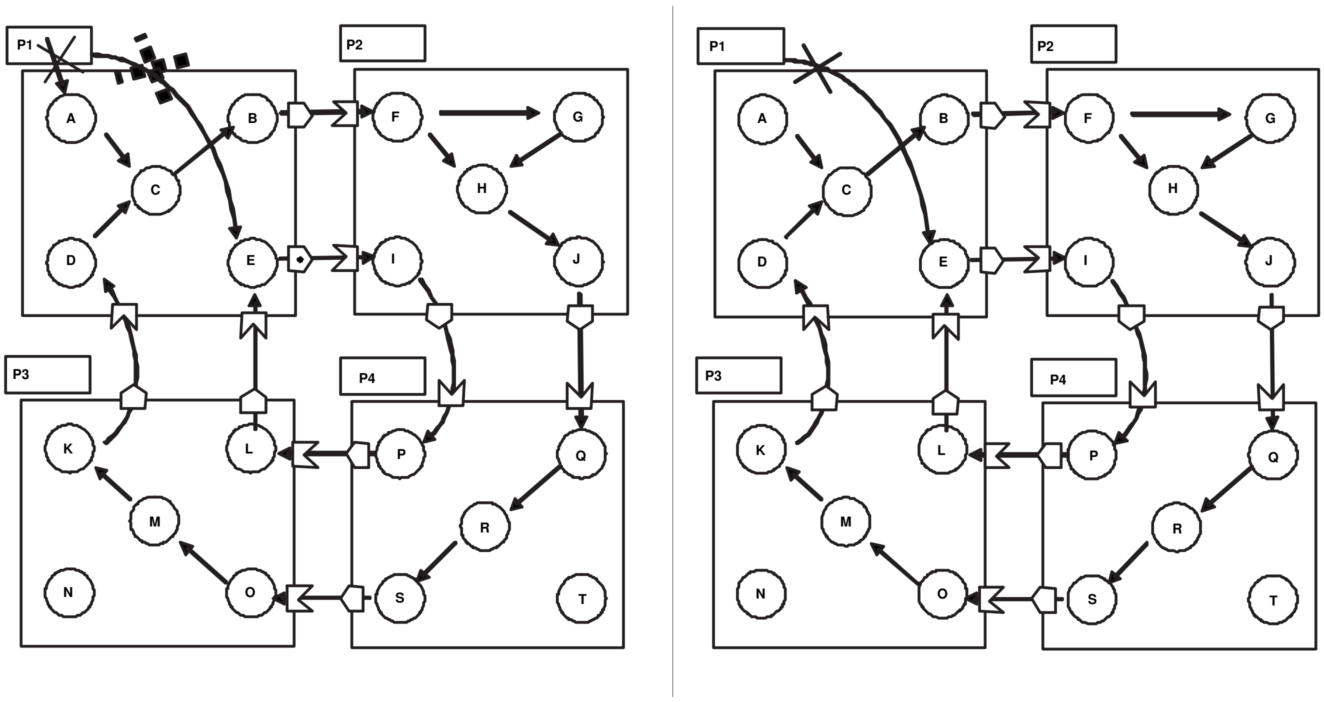


Figure III.2.6: Example B: Steps ii) and iii). The DCD receives increasingly updated information from process $P1$.

Example B: We now present an example in which the DCD has access to partial or outdated information, about one of the processes holding objects comprised in distributed cyclic garbage. In the initial situation portrayed in Figure III.2.5, the DCD has no information from $P1$ available. It constructs a DGC-consistent cut with snapshots from processes $P2..P4$. Thus, even though there are no GC local-roots in any of the processes (since the mutator in $P3$ has removed the one referencing object L_{P3}), during the CMS phase, $Scion(F_{P2})_{P2}$ and $Scion(I_{P2})_{P2}$ are promoted to the root-set of the DGC-consistent cut.

Scion promotion is performed according to the safety rules presented earlier, because the scions are targeted by stubs contained in $P1$, of which the DCD knows nothing about. Thus, conservatively, it must consider those scions as roots, to ensure safety. This is represented by the dashed grey arrows.

When the DCD performs CMS on the DGC-consistent cut, all objects become marked since they all are reachable from F_{P2} or I_{P2}. Thus, no cyclic garbage is detected, and no DeleteScion messages are issued.

After a period of time, the DCD eventually receives a snapshot from $P1$, with the following information:

- $Scion(D_{P1})_{P1} \Rightarrow \{StubsFrom \equiv \{F_{P2}\}\}$
- $Scion(E_{P1})_{P1} \Rightarrow \{StubsFrom \equiv \{I_{P2}\}\}$
- $Stub(F_{P2})_{P1} \Rightarrow \{Local.Reach \equiv false\}$, since object A_{P1} is no longer referenced from the GC

local-roots in $P1$ (the reference has been deleted by the mutator before the snapshot was taken).

- $Stub(I_{P2})_{P1} \Rightarrow \{Local.Reach \equiv true\}$, since the other reference from the GC local-roots in $P1$, targeting object E_{P1}, has been deleted by the mutator only after the snapshot in $P1$ was taken, as it is indicated by the dashed cross.

The DCD is then able to create a new DGC-consistent cut containing snapshots from processes $P1..P4$. Its root-set includes only $Stub(I_{P2})_{P1}$, marked reachable locally, in the snapshot from $P1$.

After performing the CMS, all objects belonging to the inner cycle $\{E_{P1}, I_{P2}, P_{P4}, L_{P3}\}$ are marked, while those belonging to the outer cycle are unmarked. This means the DCD is already able to detect the outer cycle but needs more up-to-date information from $P1$ (though it does not know that explicitly) in order to detect the inner cycle that is also garbage but not detectable yet.

W.r.t. the outer cycle, the DCD can issue DeleteScion messages regarding scions identified as garbage. As already mentioned, these messages may be queued and sent in batch lazily. Therefore, process $P1$ may take another snapshot after that, which still contains the same scions that were identified as cyclic garbage. Thus, when the DCD receives the new snapshot from $P1$, it is able to identify both cycles, due to the following difference in the snapshot from $P1$.

$Stub(I_{P2})_{P1} \Rightarrow \{Local.Reach \equiv false\}$, since the previous reference from the GC local-roots in $P1$ no longer exists and that fact has been registered in its snapshot.

If the DCD has recorded the DeleteScion messages issued previously, it will send DeleteScion messages only regarding the inner cycle. Once the DeleteScion messages are received by the intended processes, and the corresponding scions removed, the acyclic DGC (reference-listing) will be able to reclaim the objects comprised in the two cycles detected.

III.2.1.3 Analysis of Algorithm Properties

In this subsection, we address the relevant properties of complete distributed garbage collection, discussing them against the algorithm proposed: safety, liveness, completeness, termination, and scalability.

Safety: The DCD does not reclaim objects that may still be reachable to the mutator, due to the safety rules enforced in the CMS. The DCD is resilient and conservative w.r.t. message loss and delay (i.e., messages with graph descriptions). Replayed NewSnapshot messages cause no error as they are idempotent. Nonetheless, for simplicity, they are ignored. Messages sent earlier, and received out of order, are discarded; if considered, they would temporarily prevent algorithm progress as DGC-consistent cuts would go back in time.

Concurrency between cycle detection and the mutator must be analyzed w.r.t. two aspects: local and distributed. In local terms, snapshots are taken when the mutator is idle, or created incrementally. Nonetheless, the DCD manipulates snapshots assumed to be coherent w.r.t. each process. Regarding distributed invocations, that may swap references to objects among processes, there are no concurrency issues with the DCD; a DGC-consistent cut reveals cyclic garbage that already existed when the enclosed snapshots were created, while the mutator only manipulates live objects, which will be correctly accounted for during CMS.

Liveness: Liveness of the DCD naturally depends on processes updating their snapshots, and the DCD executing CMS on DGC-consistent cuts containing those snapshots recently received. This is guaranteed since DGC-consistent cuts are always created with the most recent version available of each snapshot. Snapshot updates need not be performed often but processes with mutator activity must eventually update their snapshots. Snapshot compression favors network usage and DCD processing.

Completeness: To achieve completeness, the DCD must eventually encompass all processes, in the sense that all of them must be included in DGC-consistent cuts. This is achieved using an hierarchical approach, by construction of higher-level DGC-consistent cuts. Thus, all processes are eventually considered, at some level, by a DCD, though not necessarily explicitly included in the snapshots sent to it.

Every snapshot received at each DCD is always included in some DGC-consistent cut, at some level. The DCD ensures that periodically, a top-level DGC-consistent cut is created that directly or indirectly spans all the processes that sent snapshots to the DCD. This higher-level cut may be sent to other DCDs it knows about.

The algorithm does not impose any pattern of cooperation among the participating processes, it is only required that all processes are eventually accounted for, directly or indirectly, in higher levels DGC-consistent cuts that may be exchanged among DCDs. Every DCD can perform at any level, simply by receiving DGC-consistent cuts from other DCDs and creating higher-level cuts, possibly exchanging them with other DCDs as well.

W.r.t. identification of garbage, even in the case where DeleteScion messages have not been issued for all scions identified as garbage in one DGC-consistent cut, and individual disjoint cycles have not been specifically identified, the algorithm still achieves completeness eventually. The remaining scions will either be removed by the acyclic DGC (when they belong to the same cycle) or will be explicitly instructed for deletion in ulterior cuts. This is so because the DCD chooses, in first place, scions for which DeleteScion messages have not been issued yet.

Termination: CMS performed on each DCD terminates since it is performed resorting exclusively to information available locally, and the termination of mark-and-sweep algorithm is trivially sound, since there is no concurrency with the mutator in the context of the DCD.

Scalability: The discussion of algorithm scalability derives from some of the arguments for completeness. The algorithm can scale to large numbers of processes because it imposes no synchronization requirements nor communication among them. There may be multiple DCDs running, possibly cooperating asynchronously in a loose hierarchy. Compressed snapshots are smaller, use limited bandwidth are only seldom sent and can be subject to CMS even if stored on disk(as in (Maheshwari and Liskov 1997c)). Snapshot compression favors network and processing.

It's worthy to note that with this information, the cycles detector does not perform a full garbage collection (as in Liskov's proposal (Liskov and Ladin 1986)). Moreover, multiple DCDs are independent and not simply replicas of a centralized detector.

The deferred nature of the DCD is exclusively used to detect cyclic distributed garbage collection. Snapshot compression and the possibility of multiple DCDs, in parallel as well as in a loose hierarchy, contribute to algorithm scalability.

The hierarchical approach does not require a strict topology (e.g., tree) and any DCD process can perform at any level of the hierarchy if it is contacted by others. There is no single top-level root of the hierarchy.

III.2.2 Algebra-based Distributed Cycle Detection

This section describes Algebra-based Distributed Cycle Detection (Veiga and Ferreira 2005a). It introduces the notion of a Cycle-Detection Algebra (**CDA**) that fully describes a detection on-course, without the need to maintain specific state in participating processes. It is used to build DGC-consistent cuts in an incremental and distributed manner, without actually exchanging process snapshots.

The algorithm detects distributed cyclic garbage ascertaining the reachability of suspected objects, selected heuristically. It employs a de-centralized approach to test whether a cycle candidate indeed belongs to cyclic garbage. Processes forward Cycle Detection Messages (**CDMs)**, containing CDA elements, that determine which reference paths are followed, in order to find if they form a distributed cycle of garbage. CDMs may be batched, and sent lazily, piggy-backed on regular acyclic DGC messages (e.g., NewSetStubs).

The cycle-detection algebra allows the representation of sub-paths already traced, and dependencies yet to be resolved, on which object reachability depends. Cycle detection messages, upon arrival, are evaluated, and matched against the snapshot of the receiving processes. As a result, detections may terminate, or a number of updated CDMs are forwarded to other processes. The combination of a CDM and the several snapshots it has been matched against, conceptually defines a DGC-consistent cut, incrementally built, that is named CDM-Graph. Several CDM-Graphs, and thus detections, may be carried out concurrently, as there is no synchronization among them. They may have processes and/or objects in common.

Intuitively, the algorithm operates as follows. It initiates cycle detection by issuing a CDM regarding a scion targeting a suspect-object. If the CDM is forwarded, across a number of processes and in the absence of mutator activity, back to the originating process with all its dependencies resolved, then, a distributed garbage cycle has been found. Therefore, the scion targeting the suspect-object can be safely deleted to break the distributed cycle.

The algorithm relies on heuristics to select candidates for cycle detection, as other suspect-based approaches (Maheshwari and Liskov 1997a; Rodrigues and Jones 1998). Contrary to these, the algorithm does not require processes to maintain state about ongoing cycle detections (Maheshwari and Liskov 1997a; Rodrigues and Jones 1998). Moreover, it does not require the collaboration of the LGC in each process, to propagate information regarding cycle detection (Louboutin and Cahill 1997; Rodrigues and Jones 1998; Fessant 2001). W.r.t. DGC-Consistent Cuts, this algorithm has the following advantages: i) cycle detection is completely de-centralized, and ii) processes need only communicate with other process they have references to.

III.2.2.1 Algorithm

Figure III.2.7 presents an overview of distributed cycle detection using an example situation analogous to the one previously presented in Section III.2.1. It comprises four processes running a distributed application. We also assume there is a pre-existing acyclic DGC algorithm deployed (e.g., reference-listing) and, thus, each process already has a DGC component running, that stores its DGC structures and, periodically, sends NewSetsStubs messages.

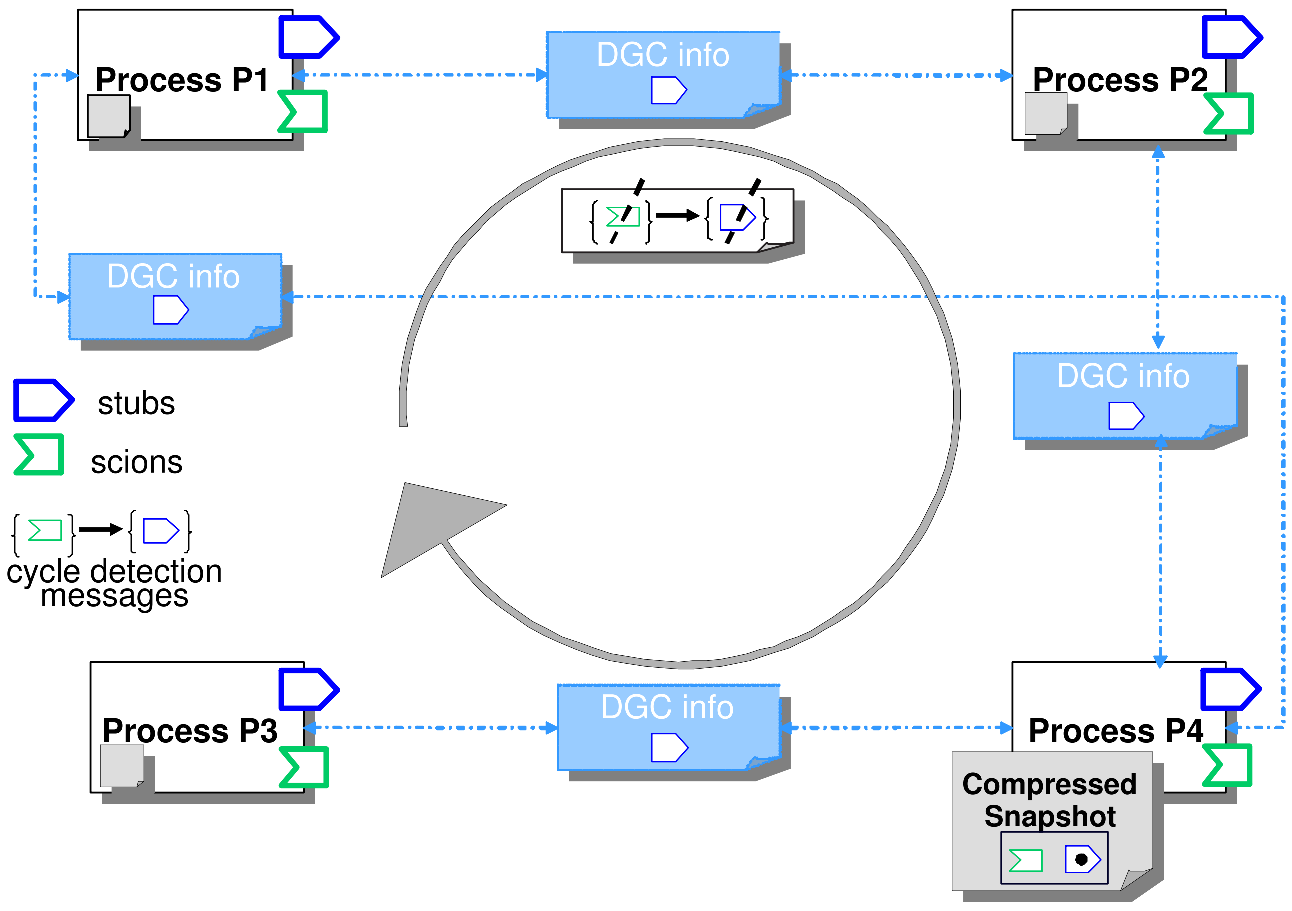


Figure III.2.7: Algorithm overview

Since reference-listing is not complete, there is a distributed cycles detector (**DCD**), for cyclic DGC. It is deployed in a de-centralized fashion. There is an instance of the DCD for each one of such processes. In this case, each DCD component manipulates just one snapshot, the one of its enclosing process, that is seldom updated. Older versions (in this case, only one) can be promptly discarded. Even though snapshots are not exchanged among processes, they are still compressed (as previously described) for increased efficiency.

The DCD in each process receives, evaluates, and possibly forwards CDMs. CDMs are sent regarding a specific scion (its destination scion), and carry information about: i) transversed stubs, and ii) dependency scions. Transversed stubs indicate a sub-path that has been traced, across snapshots, during a cycle detection. Dependency scions indicate sub-paths that must also be traced before a possible cycle can be safely detected. When a DCD receives a CDM, it evaluates the CDM and determines if a cycle has been detected. If not, it must decide whether to stop detection, or to forward the CDM.

A detection stops when the DCD discovers that : i) the sub-path being traced includes objects that are reachable from the GC local-roots of the process, or ii) there have been distributed invocations on objects comprised in the sub-path being traced, or iii) it can no longer update the CDM with new information, and a cycle has not yet been found. Either way, the DCD decides to stop that detection because either there is no garbage cycle: situations i) and ii); or it cannot progress any further with the available information, thus avoiding repeating work: situation iii).

Otherwise, the DCD decides to forward a CDM, with updated content, to every process containing the counterpart scions of the stubs reached by the DCD, when it transversed the snapshot, from the destination scion included in the CDM.

When a CDM is forwarded via a distributed cycle, transversing it completely and resolving all dependencies, without detecting any GC local-roots nor interference with the mutator, the DCD can safely determine that a distributed garbage cycle has been detected. Then, it may instruct the cyclic DGC component, running on the same process, to remove the destination scion referred in the CDM.

III.2.2.1.1 Data Structures

The DCD uses data structures already present in the acyclic DGC algorithm and defines new ones that are exclusive to cycle detection. The extensions presented in the previous section, with stubs and scions storing time-stamps, and processes keeping vector-clocks, may also be employed, but are only of use to the acyclic DGC. They are not required by this algorithm. Thus, the data structures managed by it, and their relevant extensions are:

- **Scion:** It is extended with a numeric field, an $invocation\ counter\ (IC)$, for concurrency purposes. This counter is incremented each time a remote invocation (or reply) is performed through the corresponding remote reference. Its new value is piggy-backed in the reply.
- **Stub:** It is extended with a numeric field, an $invocation\ counter\ (IC)$, for concurrency purposes. This counter is updated with the value piggy-backed in reply messages of remote invocations.
- **Snapshot**: representation of the object graph of a process, including its stubs and scions. In this algorithm, snapshots need not record the vector-clock maintained by the process, even if it is used for acyclic DGC.
- **Cycle-Detection Algebra (CDA) Element**: representation of a cycle detection in course (i.e., transversed stubs and dependency scions) that is evaluated within the context of a process, to determine whether : i) a cycle has been detected, ii) detection should stop, iii) the algebra element should be extended and the result forwarded to another process.

III.2.2.1.2 Messages

The algorithm exchanges only one kind of messages:

- **CDM - Cycle Detection Message:** message sent from the DCD component in one process to its counterpart in another process, to continue further an ongoing cycle detection. This message carries the identification of a scion in the receiving process (*destination scion*), and a CDA element that contains information already assembled about the detection.

Upon receiving a CDM, the DCD of a process ascertains if the *destination scion* is part of a distributed cycle of garbage. Otherwise, it forwards updated CDMs or stops detection.

There is no need for an explicit DeleteScion message. The DCD that ultimately detects the cycle needs only communicate with the acyclic DGC of the same process. It gives instructions to delete the *destination scion* of the CDM received. This is the scion that led to the distributed cycle being detected. Naturally, this may be implemented by exchanging an *internal* DeleteScion message between two GC components. Nonetheless, it is a message in which the sender and receiver are running on the same computer, thus no actual distributed communication is involved.

III.2.2.1.3 Cycle Detection

In this section, we describe the main idea of the DCD. We follow an intuitive description that does not consider many subtle aspects; these are addressed in the remainder of this section. However, it provides an easily understandable description of the main idea. For simplicity, we first assume that all mutators are suspended; we call this, the stop-the-world DCD. Afterwards, we do relax this requirement: concurrent DCD.

III.2.2.1.3.1 Stop-the-World DCD

Consider an hypothetical object x in process $P1$, depicted in Figure III.2.8, which is kept alive solely because it is reachable from another process, i.e. it is not locally reachable in process $P1$ (where x is allocated). If this object is not invoked for a certain amount of time we can make a guess that this object is, in fact, part of a distributed cycle of garbage. However, we are not sure about that. In order to reach a safe conclusion about x's state (live or dead), we conceived an algorithm that, intuitively, works as follows.

In process $P1$, the DCD determines which stubs (in process $P1$) are reachable from object x. Those stubs that are locally reachable (directly or indirectly from GC local-roots in $P1$) are immediately discarded from the point of view of the DCD; obviously, such stubs do not belong to a distributed cycle of garbage. On the other hand, those stubs that are solely reachable from object x, may be part of such a cycle.

Scions in process $P1$ that may also lead (directly or indirectly) to the local graph where x is included, are accounted for as extra dependencies. We define *dependency*, in cycle detection terms, as a scion that leads to the path being traced by the DCD, i.e., an alternate converging distributed path. Global reachability of the path being traced *depends*, also, on the reachability of such a scion. Therefore, if there is cyclic garbage, such a scion must also belong to it. This dependency is accounted for, and must be eventually resolved by the DCD. While it is not, no cycle has been safely identified yet. This situation is portrayed in Figure III.2.8: the remote reference from w in $P4 \rightarrow x$ in $P1$ is an extra dependency of the cycle, i.e., it is preserving the distributed cycle reachable.

So, the DCD sends a probe message along at least one of the above mentioned stubs. These probes (that are in fact, CDMs) will reach the corresponding scions in remote processes.

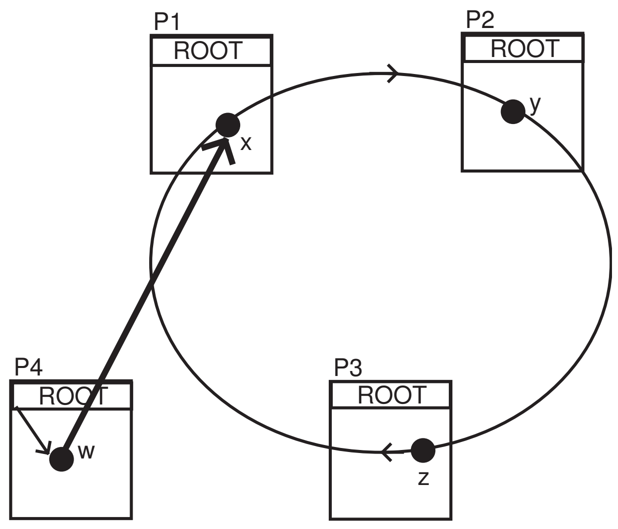


Figure III.2.8: Identifying dependencies in cycles.

For clarity, but without loss of generality, assume each process only receives one CDM. Nonetheless, different CDMs can be handled independently. Thus, in each one of such processes, the DCD performs, for each CDM received, as described next. It determines which stubs (inside the process) are reachable from the scion that received the CDM. Once again, those stubs that are locally reachable are not considered by the cycles detector; thus, the CDM does not follow the corresponding outgoing path. For those stubs that are reachable from the scion that received the CDM, and are locally unreachable, the CDM follows the corresponding outgoing path to remote processes. All other scions that may lead to any of the afore mentioned stubs, are included in the CDM, i.e., they are considered as dependencies.

Thus, the CDM is i) either stopped because, in some process, the DCD discovers a stub that is locally reachable, or ii) kept on going along the references path so that, eventually it will reach the starting process X. When such event occurs, the CDM carries an algebra that describes the distributed graph that was traversed.

This algebra may indicate that there are dependencies still to be resolved, i.e. references pointing to the graph that was traversed. In this case, it is not safe to conclude that we have discovered a distributed cycle; obviously, it is necessary to resolve such dependencies. The details w.r.t. resolving dependencies in CDA elements are presented later in Section III.2.2.1.3.3.

However, if the graph is, in fact, a distributed cycle of garbage, then it has no such dependencies yet to be resolved because all those alternate paths were fully and successfully traced. Thus, the DCD in process X, based on the algebra before mentioned, can safely conclude that it has found a distributed cycle of garbage. Therefore, all it has to do, is to delete the (local) scion from which the CDM has been initiated. Then, the distributed acyclic garbage collector will reclaim the remaining objects.

III.2.2.1.3.2 Concurrent DCD

Naturally, to assume that all mutators are suspended is not reasonable. So, periodically, each process stores a snapshot of its internal object graph on disk. This snapshot is created

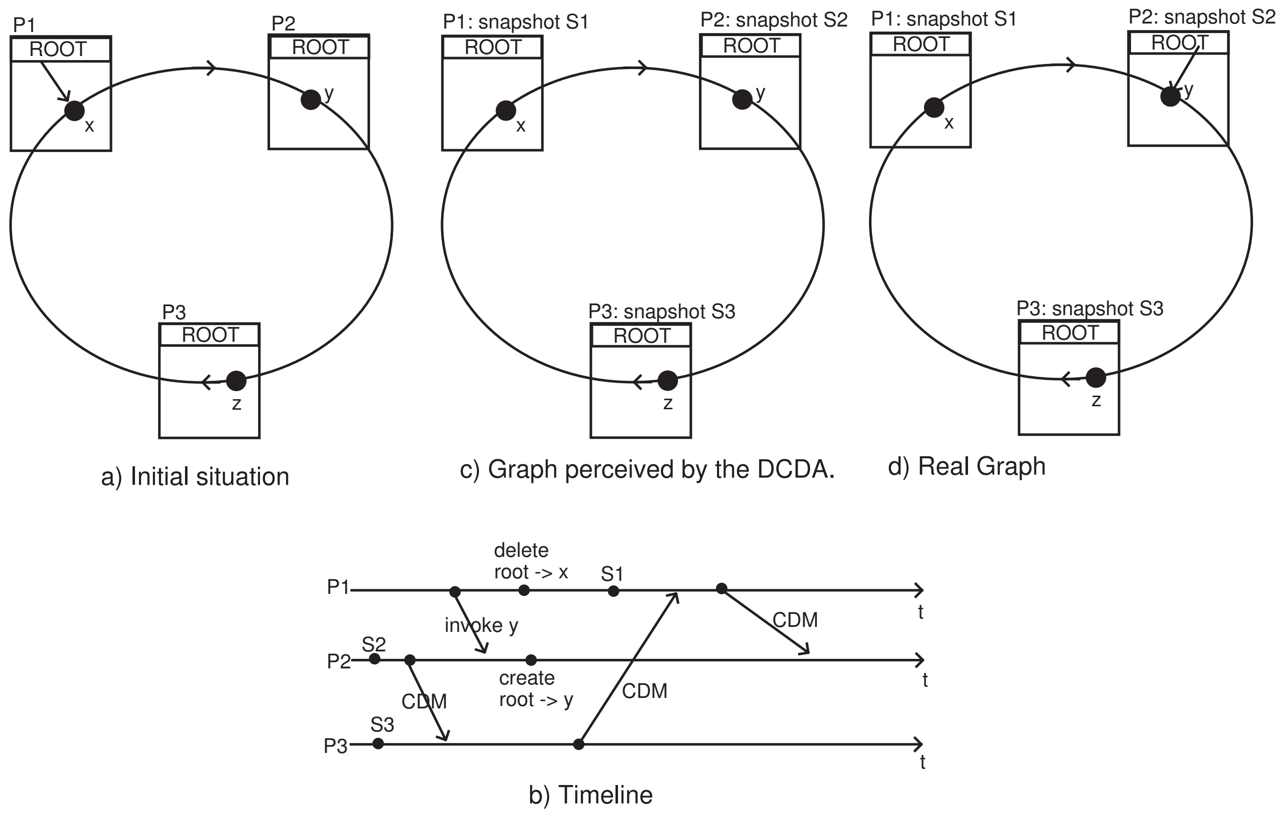


Figure III.2.9: DCD processing of independent snapshots.

by each process with no coordination w.r.t. other processes; thus, each process is completely independent.

If we assume that a set of such snapshots, taken independently by each process, provides a consistent view of the global distributed object graph, the DCD may proceed exactly as described previously. However, such an assumption is not correct. Therefore, the DCD has to ensure that the set of snapshots visited by a CDM compose, in fact, a consistent view for the purpose of finding distributed cycles of garbage.

In other words, it is only required that the sub-graph being independently traced by a CDM (to determine if it is a distributed garbage cycle) is observed consistently. This a weaker requirement than that of a consistent-cut in a distributed system due to: i) distributed cyclic garbage (as all garbage) is stable, i.e., after it becomes garbage it will not be touched again by the mutator, and ii) distributed cyclic garbage is always preserved by the acyclic DGC (that is why we need a special detector), i.e., if the DCD does nothing, it still is safe.

Thus, we define *CDM-Graph(x)* as *a GC-consistent view of a distributed sub-graph restricted as follows: including object x, and enclosed in the combination of N process snapshots.*

For DCD purposes, a CDM-Graph must respect the following invariant:*there can be no invocations along a CDM-Graph while the corresponding CDM is in transit*. If we allow this to happen, it means that the mutator is modifying the distributed graph in the back of the DCD. Consequently, the DCD may erroneously conclude that it found a garbage cycle. Fig. III.2.9-a illustrates such a case. The initial situation is that of a cycle formed by objects x, y and z in processes $P1$, $P2$ and

$P3$, respectively. This cycle is not garbage because x is referenced from the GC local-roots in $P1$.

Now consider the sequence of events depicted in the timeline in Fig. III.2.9-b ($S1$, $S2$ and $S3$ are the instants when the corresponding processes create their snapshots with no coordination at all). Suppose that the DCD starts in $P2$ by sending a CDM to $P3$. Concurrently, the mutator in $P1$, invokes y in $P2$ and deletes the reference from the GC local-root that points to x. As a result of this invocation, a new local reference is created in $P2$'s GC local-roots pointing to y. Once this invocation finishes, $P1$ creates a snapshot of its graph (instant $S1$). Given that $S2$ and $S3$ were previously taken, the view of the distributed graph that is perceived by the DCD instances (i.e., the CDM-Graph) is, in fact, the one represented in Fig. III.2.9-c, instead of the correct one represented in Fig. III.2.9-d. This would lead to the erroneous detection of a distributed cycle of garbage comprising objects x, y and z. This erroneous conclusion would be reached by the DCD if the invariant above mentioned is not respected. In this case, an invocation took place along the reference path $P1 \rightarrow P2$ that had been previously stored in the snapshot and will be included in the CDM-Graph when the CDM arrives to $P1$.

Given that snapshots are independently taken, the following situations may arise: 1) $Stub$ without corresponding $Scion$, 2) $Scion$ without corresponding $Stub$, and 3) $Stub$ with matching $Scion$. The invariant dictating the construction of a CDM-Graph is enforced using the following conservative safety rules ($Situation \Rightarrow Action$), when process snapshots are pairwise-combined through CDM:

- 1: $Stub$ without corresponding $Scion$ (snapshot of the process holding the scion is not current enough for the CDM-Graph) $\Rightarrow$ $Ignore\ CDM$.
- 2: $Scion$ without corresponding $Stub$ (reference-creation message in transit, acyclic garbage, or snapshot of the process holding the stub not current enough) $\Rightarrow$ *The CDM is never sent since there is no stub in the CDM-Graph.*
- 3a: $Stub$ with matching $Scion$ but there have been remote invocations, and possibly reference duplication, along the CDM-Graph after one of the snapshots was taken; it is not consistently accounted for in the snapshot and the CDM $\Rightarrow$ *Terminate CDM-Graph construction, i.e., terminate detection avoiding mutator-DCD race.*
- 3b: $Stub$ with corresponding $Scion$ and there were no invocations after snapshot (safe to continue CDM-Graph creation and cycle detection) $\Rightarrow$ *Proceed CDM-Graph construction, combine CDM with process snapshot and continue detection.*

III.2.2.1.3.3 Cycle-Detection Algebra

Cycle detections use an algebraic representation encoded in the CDM. The CDM content is comprised of two sets (separated by $\rightarrow$): i) a source-set holding compiled dependencies and, ii) a target-set holding target objects that the message has been forwarded to so far.

For each CDM delivered to a process, the DCD performs an algebraic matching (eliminating elements present in both sets of the CDM), thus resolving dependencies and reducing the CDA element in the message. This allows the DCD to determine whether a distributed garbage cycle

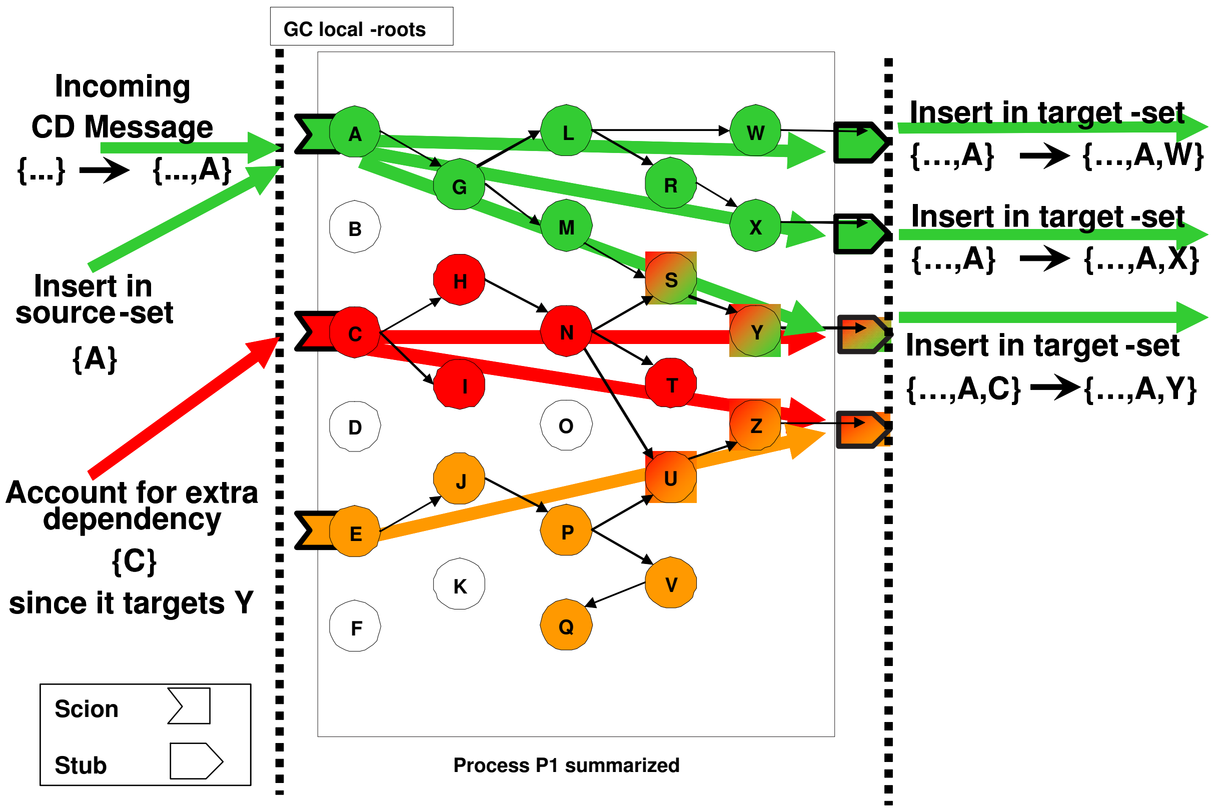


Figure III.2.10: CDM Matching and Forwarding.

has been detected. A cycle is detected when, after the algebraic matching, the source-set is reduced to an empty set. CDM matching also allows to find unresolved dependencies. These are any unmatched elements left in the source-set.

In the situation portrayed by Fig. III.2.10, the DCD in process $P1$ received a CDM with $Scion(A_{P1})_{P1}$ as its destination scion. We assume that no cycle has been detected. Therefore, $P1$ should forward the CDM along the sub-paths reachable from object A_{P1}.

Without loss of generality, and since only one process is depicted in the figure, we identify stubs by the object that contains the corresponding out-going inter-process reference (instead of the referenced object in another process). For simplicity, we assume they all target different objects, possibly in different processes. The objects containing out-going inter-process references, that are reachable from object A, are W, X, and Y. None of them is locally reachable. Therefore, according to the previously stated rules, an updated CDM should be forwarded along each of them. Nonetheless, the forwarded CDMs (in this case, three) need not be sent immediately nor simultaneously. They may be handled independently.

W.r.t the stub corresponding to the out-going remote reference contained in W, the CDA element is updated simply by inserting the stub in the target-set. It now contains A and W, and this is included in the CDM being forwarded along this stub (i.e. containing its corresponding scion as the destination-scion of the CDM). Similar actions are performed w.r.t. the stub regarding the reference contained in X.

The stub regarding the out-going reference in object Y requires an additional update to the

CDA element, in the CDM being forwarded through it, to account for the extra-dependency it adds to cycle detection. As a matter of fact, since object Y is also reachable from object C, if A and Y belong to distributed cyclic garbage, so thus must also belong object C. This means that the cycle being detected also *depends* on C belonging to it. Therefore, C is added as an extra dependency, and inserted in the source-set of the CDA element, included in the CDM being forwarded along the stub. Obviously, as in the previous cases, Y is included in the target-set, since it corresponds to the stub forwarding the CDM.

III.2.2.1.3.4 Detecting Mutator-DCD Concurrency

Interaction between the mutator and the DCD is very limited. Cycle detection is performed resorting to off-line, summarized descriptions of the memory graph in each process. Thus, there is no contention between the mutator and the DCD. Mutator actions are not delayed due to synchronization with cycle detection activity.

As it was mentioned in Section III.2.2.1.3.2, just combining independently taken graph snapshots, at every process, does not produce a consistent view of the distributed graph. However, these snapshots of different processes are pair-wise combined (with arriving CDMs) just for the purpose of detecting distributed cycles. Therefore, we do not require a global consistent view (e.g., one that is produced by a causal cut) of the distributed graph.

For each detection in course, we need to ensure that every mutator event, performed on the CDM-Graph, is completely represented in the set of snapshots (one for every process comprising the cycle). For instance, a distributed invocation involving two processes included in the CDM-Graph must be represented in both their snapshots. This stems from the fundamental property that: if a mutator has accessed the CDM-Graph *after* snapshots were created on any of the processes, it means that the cycle is not garbage, at least not yet (as far as we can tell from the information provided in the snapshots). If it is indeed a cycle, to be safe, we may need to update the snapshots of one or more of the processes. In summary, the CDM-Graph cannot be touched while detection is in course.

A distributed race between the mutator and the DCD can occur when the following sequence of events takes place:

1. There is a GC local-root in one process $P1$, targeting object x, therefore holding the cycle reachable (so no actual garbage cycle exists).

2. A detection that will reach object x is already in course but has not yet reached process $P1$.

3. The mutator performs a remote invocation (possibly chained through various processes) that switches the reachability from GC local-roots, instead of object x in $P1$, to another object in one of the processes already visited by the detection.

4. New snapshot information becomes available at $P1$, now stating that object x is no longer reachable locally.

5. When it reaches $P1$, the ongoing cycle detection will be able to trace, lazily, the whole cycle without finding any GC local-root, thus, wrongly detecting a non-existing garbage cycle.

Therefore, algorithm correctness in the presence of mutator activity lies in the ability of determining if there has been mutator activity accessing the cycle itself. This is a natural reason to abort cycle detection. However, safety must be preserved without incurring the mutator in significant delays.

The word *after*, employed previously, does not imply any notion of <<global timing>>; just causality between each pair of processes (determined by mutator events and restricted to the CDM-Graph), solely for the purpose of each cycle detection. A particular case of this situation happens when, for instance, a CDM is delivered to a scion that is not yet inscribed in the snapshot (it was created after the last snapshot was taken). In this case, this CDM is simply discarded and the detection terminated.

There are two straightforward ways to uphold the CDM-Graph invariant w.r.t. the last two rules (i.e., rules 3a and 3b) presented earlier: i) *pessimistic*: to freeze the mutator, or deny access to the path already transversed while detection is in course, or ii) *optimistic*: to detect, at a later stage, that this invocation has indeed occurred.[2] The first option is clearly undesirable as it disrupts applications with no justification (if the mutator wants to access objects, they are clearly not garbage). The second option allows the mutator to run at full-speed at the expense of possibly wasting some detection work (an hypothetical distributed cycle may be partially or completely transversed by the detector, only to find out that meanwhile, a distributed invocation on that cycle has taken place). This is achieved by verification of invocation-counter fields contained in CDMs and process snapshots.

The algorithm needs only to ensure safety in these cases (and it does) since they must be infrequent when efficient heuristics are used to select cycle candidates. Thus, the solution conceived consists on a barrier that detects invocations being performed in the back of the DCD.

A cycle detection starts assuming that the objects traced are, indeed, a cycle. If during the flow of a CDM across a series of processes, those objects are invoked/accessed, there should be a way to revise this assumption in the event of this new information. When compared to local GC, a barrier (Wilson 1992) implies that an object modified by the mutator, after it was traced, must be re-scanned to account for *new* descendants previously considered as garbage. Thus LGC must correctly identify all live objects. In the case of the DCD, given that it identifies cyclic garbage that is kept "alive" by the acyclic DGC, the safe thing to do is costless: just do nothing. Thus, the DCD barrier uses a simple invocation-counter (described in Section III.2.2.1.1) that allows the DCD at $P2$ to detect that an invocation has taken place. Thus, the cycle under consideration is not considered to be garbage. The use of invocation-counters is exemplified in Section III.2.2.2.3.

III.2.2.1.4 Optimizations

The algorithm presented can be improved by three classes of optimizations: i) snapshot compression, ii) CDM queueing for ulterior combination, and iii) restricting CDM propagation. These optimizations do not compromise neither the safety of the algorithm, nor the eventual detection of all distributed cycles (i.e., completeness).

[2]With assembly-language synchronization primitives, this would correspond to using LOCK or LOAD-LINK/STORE-CONDITIONAL, respectively.

Snapshot Compression: As described in the previous chapter, snapshots of processes may be very large. Snapshot compression, w.r.t. Algebra-based Cycle Detection, is not required for network efficiency because snapshots are never exchanged among DCD in different processes. Still, to save storage, and improve efficiency, snapshots are compressed via graph summarization. As in the previous algorithm, graph summarization omits all object scalar content and references strictly internal to the process. The DCD only uses snapshots in their compressed form.

Information about reachability associations, among stubs and scions in each process, is explicitly represented in bidirectional manner. There is a $StubsFrom$ list for each scion, as in the previous algorithm, as well as a $ScionsTo$ list for each stub.

The $StubsFrom$ list, for each scion, allows the DCD to determine, while detecting a cycle, the next set of processes (targeted by out-going references) that should be probed (by the CDM) in order to transverse the full cycle.

On the other hand, the $ScionsTo$ list, in each stub, allows the DCD to determine extra dependencies that must be also verified before the cycle is correctly identified. This extended information contained in the compressed snapshot avoids the need to perform back-tracing, from stubs to scions, which would be complex (requiring additional information in objects), time-consuming, and repetitious.

Finally, the $Local.Reach$ flag, in each stub, indicates the local reachability of the stub.[3] This ensures that a cycle comprising an object which is reachable from the GC local-roots, of its enclosing process, is never wrongly identified as garbage. When such an object is found, cycle detection along that path is terminated, with a negative result w.r.t. cycle detection.

Optimizing CDMs: CDM forwarding can be subject to optimization, using a best effort approach. When the reception of a CDM results in a number of updated CDMs being forwarded, the DCD may queue some of them (e.g., all but one), and delay their forwarding for a period of time (e.g., a time-out, or until the queue dimension reaches a threshold value). If the forwarded CDMs are not transversing mutually-referenced garbage cycles, the CDMs in queue will eventually be forwarded, only later, which does not hinder completeness.

If, however, there are mutually-referenced cycles, some of the forwarded CDMs will eventually arrive back at the process where the previous ones were queued. If there have been no changes to the process snapshot, the number of CDMs to be forwarded will be the same as before. This time, though, there are already CDMs queued in the process, with the same destination scions. Queued CDMs with identical destination scions may be subject to suppression or combination.

Suppression consists in dropping a CDM because another one, with the same destination-scion, has strictly more information (i.e., the source and target-sets of the latter are strict supersets of those of the former). The former is deemed obsolete and hence needs not be forwarded. The dropped CDM is usually the one that was queued, although this may not be the case if several detections of the same cycle start with different candidates.

[3]In fact, local reachability of, at least, one of the objects containing the corresponding out-going inter-process reference.

CDMs with the same destination scion may also be combined, i.e., the source and target-sets of the resulting CDM are obtained from the union, respectively, of the source and target-sets of the individual CDMs. Suppression is actually a special form of combination, where the resulting CDM is identical to one of the CDMs that were combined.

Group Awareness: If processes are organized in groups (e.g., one group containing processes in the same LAN), the DCD may queue and delay forwarding CDMs destined to processes outside the group (that would be expensive), and forward CDMs destined to members of the group immediately (that are cheaper). This reduces the number of CDMs sent outside the group, that in turn, result from the combination of several CDMs forwarded within the group.

III.2.2.2 Prototypical Example

In this section we present a number of prototypical situations that exemplify, and further explain the main aspects of the algorithm presented. The examples follow the same presentation order of the algorithm description. The first two examples are explained assuming the mutator is stopped in all processes. This allows a description easier to understand. Issues regarding concurrency with mutators are described in the third example.

For clarity, we use simplified language for certain expressions and aspects, when there is no danger of confusion. In particular, objects are represented by their name (a letter) and their enclosing process (e.g., A_{P1}, see Fig.III.2.11). Sub-graphs of connected objects may be represented in abbreviation (e.g., $\{\{A,\ C,\ B\}_{P1}, \{F,\ G,\ H\}_{P2}\}$), aggregated by its/their enclosing process. References may be also explicitly described when relevant (e.g., $B_{P1} \rightarrow\ F_{P2}$).

Once again, throughout the examples, for clarity, when we say that: i) *stubs are reachable from a scion*, or that ii) *scions that lead to a stub*, we are using simplified language. Actually, we mean: i) *stubs accounting for out-going references enclosed in objects, that are reachable from a specific object, targeted by an incoming remote reference*, or ii) *scions, accounting for incoming remote references, whose target objects point, directly or indirectly, to objects holding out-going remote references to a specific remote object, represented by a stub*, respectively.

For example, w.r.t Figure III.2.11 addressed in detail in Section III.2.2.2.1, we would have the following information, maintained in process $P2$, regarding object F: $StubsFrom(F_{P2}) = \{Q_{P4}\}$, i.e., the stub in $P2$ that represents the remote-reference to Q_{P4} is reachable ***from*** the scion targeting object F_{P2}. This implies that the scion targeting object F_{P2} ***leads to*** the afore mentioned stub and therefore the following information is also maintained at $P2$: $ScionsTo(Q_{P4}) = \{F_{P2}\}$ (more generally, we would have that $ScionsTo(Q_{P4}) \supseteq \{F_{P2}\}$). The previous examples use a simplified notation in order to be more intuitive. Nonetheless, we will use a more formal notation in the remaining examples.

III.2.2.2.1 Example A: Detecting a Simple Cycle

The first example portrays a a simple situation, shown in Fig. III.2.11. There are four processes involved: $P1$ through $P4$. Remote references (e.g., $B_{P1} \rightarrow F_{P2}$) are represented with their

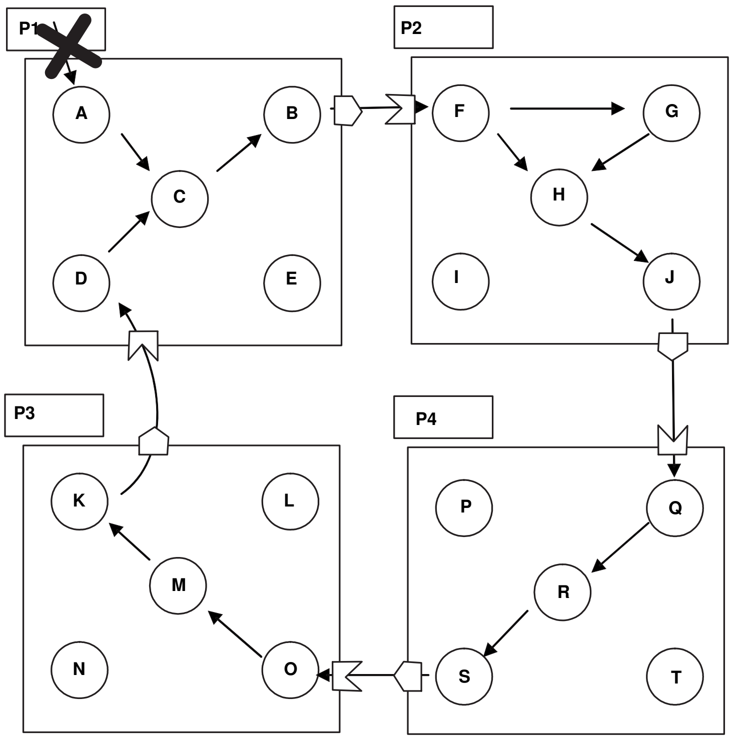


Figure III.2.11: A simple distributed garbage cycle.

associated stubs (e.g., at B_{P1}) and scions (e.g., at F_{P2}). There is a distributed garbage cycle since object A_{P1} has ceased to be reachable from the GC local-roots in $P1$, and $P1$ has subsequently taken a snapshot. As there are no other reachability roots, the whole cycle is garbage, yet undetectable by acyclic DGC. The cycle can be represented by the following chain of objects (starting and finishing in $P2$):

$$\{\{F, G, H, J\}_{P2}, \{Q, R, S\}_{P4}, \{O, M, K\}_{P3}, \{D, C, B\}_{P1}\}$$

In the example, the compressed snapshot at process $P2$ holds the following data (symbol $\Rightarrow$ means $evaluates\ to\ or\ returns$, $\equiv$ relates a field name and its value):

$$Scion(F_{P2})_{P2} \Rightarrow \{StubsFrom \equiv \{Q_{P4}\}\}$$

$$Stub(Q_{P4})_{P2} \Rightarrow \{ScionsTo \equiv \{F_{P2}\}, Local.Reach \equiv false\}$$

This means that, in $P2$: i) $Stub(Q_{P4})$ is reachable *from* $Scion(F_{P2})$, ii) $Scion(F_{P2})$ leads *to* $Stub(Q_{P4})$, and iii) $Stub(Q_{P4})$ is not reachable from the GC local-roots of $P2$.

Consider that a cycle detection is initiated with $Scion(F_{P2})_{P2}$ as a candidate. We describe now how the CDA elements (source-set and target-set) encoded in the CDM evolve, as the cycle detection detection progresses. The steps performed and relevant state are the following:

1. $P2 : Alg_0 \Rightarrow \{\{F_{P2}\} \rightarrow \{\}\}$, ($F_{P2}$ chosen as candidate for cycle detection; it is the first dependency)
2. $P2 : StubsFrom(F_{P2}) \Rightarrow \{Q_{P4}\}$, (stubs reachable from F_{P2}).

3. $P2$: $Alg_1 \Rightarrow \{\{F_{P2}\} \rightarrow \{Q_{P4}\}\}$, (result algebra after including new found stub; scion already included).
4. $P2$: $Send\ Alg_1\ to\ P4$, (send CDM to process $P4$).
5. $P4$: $Deliver\ Alg1$
6. $P4$: $Algebra\ Matching\ for\ Alg_1$

In the previous step (#6), matching of source ($\{F_{P2}\}$) and target ($\{Q_{P4}\}$) sets in the message would produce, as expected, the following result:

6. $P4$: $Matching(Alg_1) \Rightarrow \{\{F_{P2}\} \rightarrow \{Q_{P4}\}\}$
7. $P4$: $Cycle\ Found \Rightarrow\ false$

This is due to the fact there are no intersecting elements in the two sets, therefore, no matching could be performed.

Following the flow of the CDM, consider now that similar steps (1...4) were performed, this time, at $P4$. They would render the following result:

8. $P4$: $Alg_1 \Rightarrow \{\{F_{P2}\} \rightarrow \{Q_{P4}\}\}$
9. $P4$: $StubsFrom(Q_{P4}) \Rightarrow \{O_{P3}\}$
10. $P4$: $Alg_2 \Rightarrow \{\{F_{P2}, Q_{P4}\} \rightarrow \{Q_{P4}, O_{P3}\}\}$
11. $P4$: $Send\ Alg_2\ to\ P3$

And, once the CDM carrying Alg_2 arrives at P3:

12. $P3$: $Deliver\ Alg2$
13. $P3$: $Matching(Alg_2) \Rightarrow \{\{F_{P2}\} \rightarrow \{O_{P3}\}\}$
14. $P3$: $Cycle\ Found \Rightarrow\ false$

This result shows, after matching, the relevant information for cycle detection. It illustrates that, until this point, cycle detection relies on establishing a path that eliminates dependency on F_{P2}, and that the wave front of detection is placed on O_{P3}. This agrees with the intuitive result that, starting in $P2$, no cycle can be safely detected in the path $P2 \rightarrow P4 \rightarrow P3$. Continuing detection at $P3$:

15. $P3$: $StubsFrom(O_{P3}) \Rightarrow \{D_{P1}\}$
16. $P3$: $Alg_3 \Rightarrow \{\{F_{P2}, Q_{P4}, O_{P3}\} \rightarrow \{Q_{P4}, O_{P3}, D_{P1}\}\}$
17. $P3$: $Send\ Alg_3\ to\ P1$

Upon CDM arrival at $P1$:

18. $P1$: $Deliver\ Alg3$
19. $P1$: $Matching(Alg_3) \Rightarrow \{\{F_{P2}\} \rightarrow \{D_{P1}\}\}$

20. $P1 : Cycle\ Found \Rightarrow\ false$

And, preparing CDM for forwarding:

21. $P1 : StubsFrom(D_{P1}) \Rightarrow \{F_{P2}\}$
22. $P1 : Alg_4 \Rightarrow \{\{F_{P2}, Q_{P4}, O_{P3}, D_{P1}\} \rightarrow \{Q_{P4}, O_{P3}, D_{P1}, F_{P2}\}\}$
23. $P1 : Send\ Alg_4\ to\ P2$

When the CDM arrives at process $P2$:

24. $P2 : Deliver\ Alg4$
25. $P2 : Matching(Alg_4) \Rightarrow \{\ \{\} \rightarrow \{\}\ \}$
26. $P2 : Cycle\ Found \Rightarrow\ true$

At this moment, for the DCD, it is safe to assume that a cycle has been found and that object F_{P2} belongs to it. Therefore, it is safe to instruct the acyclic DGC at $P2$ to **delete the scion** accounting for the remote reference to F_{P2}. Later, when the LGC runs on process $P2$, the stub accounting for remote reference to Q_{P4} will disappear. This will in sequence, after LGCs in each process, reclaim the distributed cycle. Once again, in the previous steps, were any of the objects reachable locally (in its enclosing process), that fact would be reflected upon the reachability bit-flag in one or more stubs, and cause termination of the cycle detection.

III.2.2.2.2 Example B: Detecting Mutually Referenced Cycles

An important prototypical example of complex cyclic garbage is that of mutually-referenced cycles (see Figure III.2.12). We describe how the DCD detects such cycles. Obvious steps, similar to those in the previous example, are omitted for simplicity.

Let us assume that detection starts, once again, at object F_{P2}. Then we have:

1. $P2 : StubsFrom(F_{P2}) \Rightarrow\ \{V_{P5},\ K_{P3}\}$
2. $P2 : Alg_{1a} \Rightarrow\ \{\{F_{P2}\} \rightarrow \{V_{P5}\}\}\ and\ send\ to\ P5$
3. $P2 : Alg_{1b} \Rightarrow\ \{\{F_{P2}\} \rightarrow \{K_{P3}\}\}\ and\ send\ to\ P3$

Since $StubsFrom(F_{P2})$ has more than one element, several different CDM derivations are created with different out-going paths (in this case: one regarding V_{P5} and other to K_{P3}).

Upon arrival of CDM carrying Alg_{1a} at $P5$ (we address Alg_{1b} in the end of the example):

4. $P5 : StubsFrom(V_{P5}) \Rightarrow\ \{T_{P4}\}$
5. $P5 : ScionsTo(\{T_{P4}\}) \Rightarrow\ \{Y_{P5}\}$
6. $P5 : Alg_{2a} \Rightarrow \{\{F_{P2},\ V_{P5},\ Y_{P5}\} \rightarrow \{V_{P5},\ T_{P4}\}\},\ and\ send\ to\ P4$

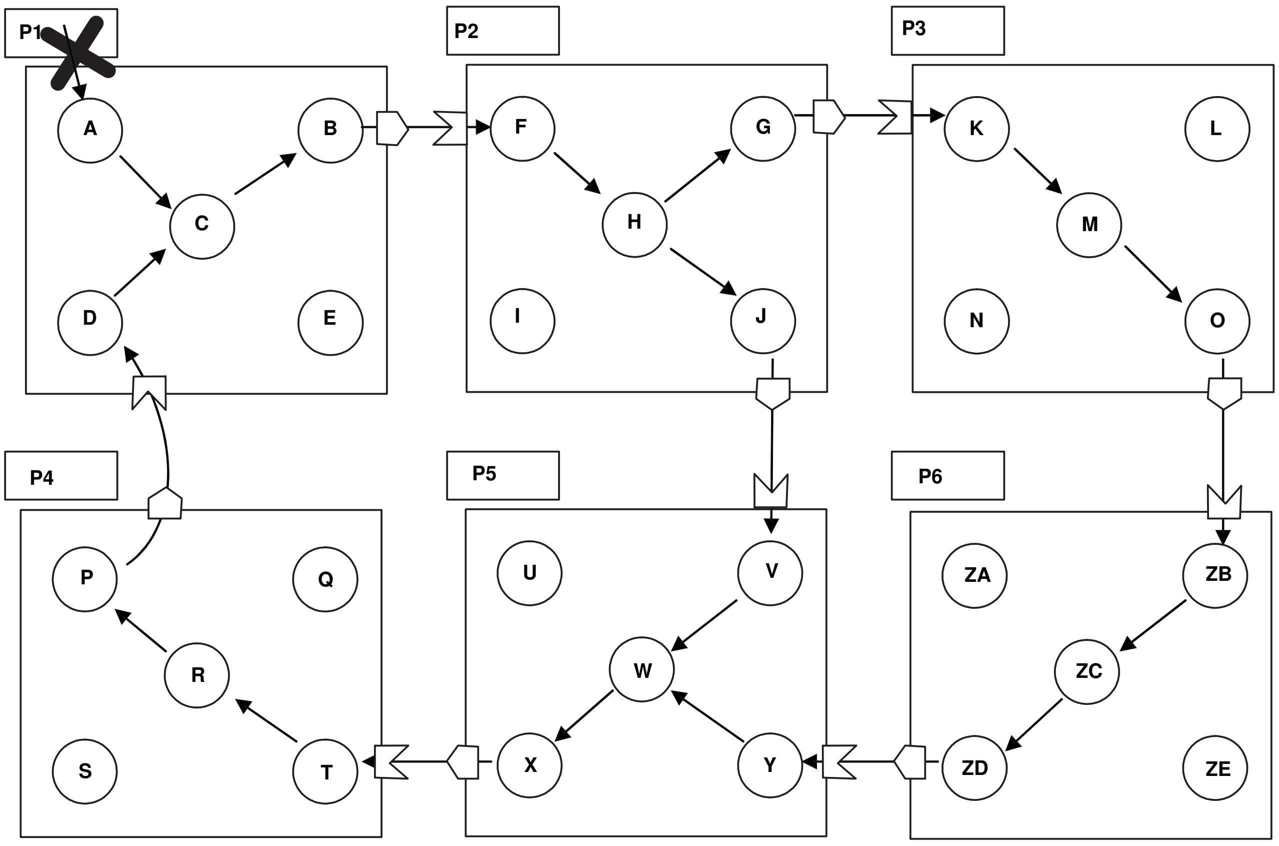


Figure III.2.12: Two mutually-linked distributed garbage cycles.

Note that on step #5, an additional pass is performed. For every stub reachable from the given scion, all other scions that may lead to the same stub (in this case $Y_{P5} \rightarrow T_{P4}$), are also accounted for as dependencies to be resolved. This information is readily available in the summarized graph description in $P5$. In Section III.2.2.2.1, accounting for extra dependencies was omitted. It would be redundant since there was an one-to-one correspondence between stubs and scions (e.g., in step #2 of Example A: $ScionsTo(StubsFrom((F_{P4})_{P2}) \Rightarrow \{F_{P4}\})$ and, therefore, no extra dependencies were added.

Upon arrival at $P4$ and applying the algorithm, the outcome includes the following results at each process:

7. $P4: \ Alg_{3a} \Rightarrow \{\{F_{P2},\ V_{P5},\ Y_{P5},\ T_{P4}\} \rightarrow \{V_{P5},\ T_{P4},\ D_{P1}\}\},\ send\ to\ P1$
8. $P1: \ Alg_{4a} \Rightarrow \{\{F_{P2},\ V_{P5},\ Y_{P5},\ T_{P4},\ D_{P1}\} \rightarrow \{V_{P5},\ T_{P4},\ D_{P1},\ F_{P2}\}\},\ send\ to\ P2$

When the CDM arrives at $P2$, one of the cycles has been transversed. Detection continuing at $P2$ performs the following steps:

9. $P2: \ Deliver\ Alg_{4a}$
10. $P2: \ Matching(Alg_{4a}) \Rightarrow \{\{Y_{P5}\} \rightarrow \ \{\}\}\}$
11. $P2: \ Cycle\ Found \Rightarrow \ false$

Naturally, the algorithm is not able to infer that a cycle has been found. Moreover, matching of Alg_{4a} states that there still is an unresolved dependency on Y_{P5}. This agrees with Fig. III.2.12 where the reference $ZD_{P6} \rightarrow \; Y_{P5}$ represents a branch of the rightmost cycle connecting with the leftmost cycle. Cycle detection will proceed as presented next:

12. $P2: \; StubsFrom(F_{P2}) \Rightarrow \; \{K_{P3}, \; V_{P5}\}$
13. $P2: \; Alg_{5a,a} \Rightarrow \{\{F_{P2}, \; V_{P5}, \; Y_{P5}, \; T_{P4}, \; D_{P1}\} \rightarrow \{V_{P5}, \; T_{P4}, \; D_{P1}, \; F_{P2}, \; K_{P3}\}\}, \; send\,to\,P3$
14. $P2: \; Alg_{5a,b} \Rightarrow \{\{F_{P2}, \; V_{P5}, \; Y_{P5}, \; T_{P4}, \; D_{P1}\} \rightarrow \{V_{P5}, \; T_{P4}, \; D_{P1}, \; F_{P2}\}\}, \; send\,to\,P5$
15. $P2: (\; Alg_{4a} = \; Alg_{5a,b}) \Rightarrow \; true, \; stop\,CDM\;forwarding\;for\;Alg_{5a,a}, terminate\;branch$

In the previous steps, in process $P2$, two different derivations of Alg_{4a} were created. The first one, $Alg_{5a,a}$, created due to stub $(K_{P3})_{P2}$ should be forwarded to $P3$. Regarding $Alg_{5a,b}$, no forwarding should occur and this branch of detection should be terminated. This stems from the fact that this CDM derivation holds information about a cycle, the leftmost in Figure III.2.12, that has already been traced, and that would be traced again if $Alg_{5a,b}$ was forwarded to $P5$.

Thus, no new information was obtained and there is no point in continuing. If not, it would loop forever with the same outcome, i.e., denouncing a dependency of the leftmost cycle on Y_{P5}. This ensures algorithm termination w.r.t cyclic garbage whose reachability is dependent of upstream acyclic garbage not yet reclaimed by the acyclic DGC. However, this may not always be the case. If there are alternate out-going paths emerging from the cycle, these should receive a CDM with information about the complete inner cycle. This issue is described in greater detail in Section III.2.2.3, when algorithm termination is analyzed.

Upon arrival of $Alg_{5a,a}$ at $P3$ we have:

16. $P3: \; Deliver\; Alg_{5a,a}$
17. $P3: \; Matching(Alg_{5a,a}) \Rightarrow \{\{Y_{P5}\} \rightarrow \{K_{P3}\}\}$
18. $P3: \; Cycle\;Found \Rightarrow \; false$

Preparing the next CDM to forward:

19. $P3: \; StubsFrom(K_{P3}) \Rightarrow \{ZB_{P6}\}$
20. $P3: \; Alg_{6a,a} \Rightarrow$
    $\{\{F_{P2}, \; V_{P5}, \; Y_{P5}, \; T_{P4}, \; D_{P1}, \; K_{P3}\} \rightarrow \{V_{P5}, \; T_{P4}, \; D_{P1}, \; F_{P2}, \; K_{P3}, \; ZB_{P6}\}\}, \; send\,P6$

Upon arrival of $Alg_{6a,a}$ at $P6$ we have, this time abbreviated:

21. $P6: \; Matching(Alg_{6a,a}) \Rightarrow \{\{Y_{P5}\} \rightarrow \{ZB_{P6}\}\}$
22. $P6: \; Cycle\;Found \Rightarrow \; false$
23. $P6: \; StubsFrom(ZB_{P6}) \Rightarrow \{Y_{P5}\}$
24. $P6: \; Alg_{7a,a} \Rightarrow$
    $\{\{F_{P2}, \; V_{P5}, \; Y_{P5}, \; T_{P4}, \; D_{P1}, \; K_{P3}, \; ZB_{P6}\} \rightarrow \{V_{P5}, \; T_{P4}, \; D_{P1}, \; F_{P2}, \; K_{P3}, \; ZB_{P6}, \; Y_{P5}\}\}, \; send\,P5$

And, upon arrival of $Alg_{7a,a}$ at $P5$ we have, finally:

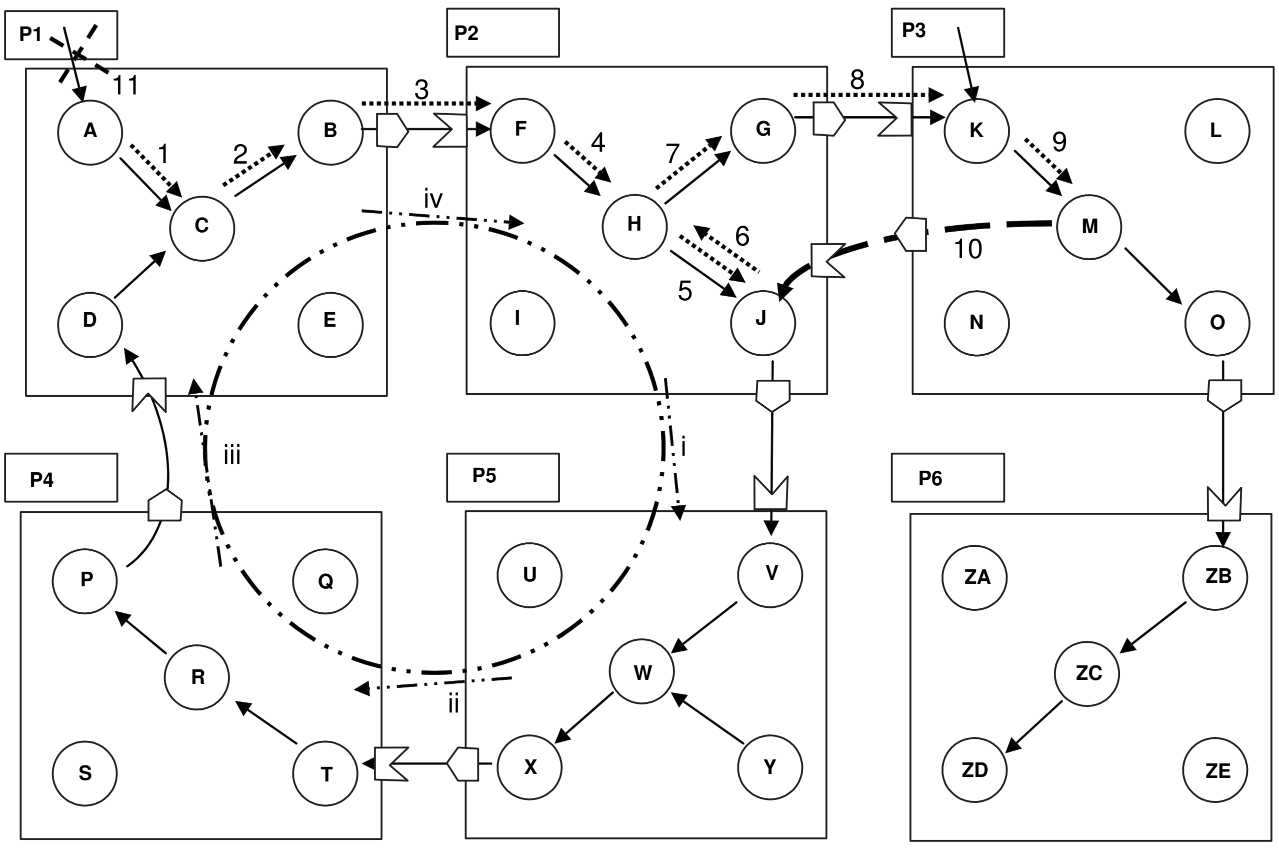


Figure III.2.13: A race between mutator and cycle detection.

25. $P5: Matching(Alg_{7a,a}) \Rightarrow \{\{\} \rightarrow \{\}\}$

26. $P5: Cycle\ Found \Rightarrow\ true$

The distributed mutually referring cycles could have also been detected if derivation Alg_{1b} (see step 3) had been continued.

Several detections can be performed in parallel, at any rate of progress, and comprising any number of processes, without conflict. Other situations mixing acyclic garbage (either upstream or downstream) with cyclic garbage are also solved with the cooperation of the acyclic DGC (prior to cycle detection and after, respectively).

III.2.2.2.3 Example C: Detecting a Mutator-DCD Race

To explain the behavior of the algorithm, in presence of concurrency due to distributed invocations (that may cause creation of inter-process references), we use the example depicted in Figure III.2.13. There are six processes ($P1...P6$). There are two independent sequences of events: i) mutator-caused events (numbered 1...11), and ii) cycle detection events (numbered $i...iv$). Algorithm safety consists in showing correct behavior in spite of any interleave of the two sequences. We assume, for simplicity, that there are updated snapshots, in every process, and available before event 1 and event i:

If no more snapshots are taken, present information is sufficient in order to handle the situation safely: when CDM arrives at P1 (instant iii, regardless of mutator activity), the DCD will be informed that $Local.Reach(B_{P2}) \Rightarrow\ true$ and will abort detection.

However, if the compressed snapshot in $P1$ is updated after event 11 (root erasure), and before event iii, with $11 \prec\ iii$, there is a problem, described next. The combination of graph information at $P1$, and cycle detection information forwarded from $P2 ... P5 ... P4$ in iii, will produce an inconsistent view of the distributed graph. Upon arrival of the CDM at $P1$, in this case, we have $Local.Reach(B_{P2}) \Rightarrow\ \ false$ and, after algebra matching, the DCD will forward this information in a CDM to $P2$ where the cycle will, eventually, be erroneously detected (since it is now kept live by the GC local-root in $P3$).

In order to prevent this inconsistent behavior, the DCD must be informed of mutator activity in any part of the path it will trace. We explain, now, how this race condition is avoided.

Recalling the definition of the algorithm's data-structures, there is an additional invocation-counter (IC) associated with every stub and scion. This extra field IC, included in every stub and scion, is incremented and piggy-backed, each time a remote invocation (or reply) is performed through the corresponding remote reference.

In the previous example, let us assume that we have, before events 1 and i as DGC info:

$Stub(F_{P2})_{P1} \Rightarrow \{IC \equiv\ x\}$

$Scion(F_{P2})_{P2} \Rightarrow \{IC \equiv\ x\}$

And, off-line, a compressed snapshot for cycle detection:

$Stub(F_{P2})_{P1} \Rightarrow \{ScionsTo \equiv \{D_{P1}\}, Local.Reach \equiv\ true, IC \equiv\ x\}$

$Scion(F_{P2})_{P2} \Rightarrow \{StubsFrom \equiv \{Q_{P4}\}, IC \equiv\ x\}$

Consider, again, the sequence of events presented where race conditions can occur. CDMs hold invocation-counters, only when they are relevant to this race condition, others are omitted:

1. $t =\ i@P2 : Alg_{1a} \Rightarrow\ \{\{\{F_{P2}, x\}\} \rightarrow \{V_{P5}\}\},\ and\ send\ to\ P5$
2. $t =\ ii@P5 : Alg_{2a} \Rightarrow\ \{\{\{F_{P2}, x\}, V_{P5}\} \rightarrow \{V_{P5}, T_{P4}\}\},\ and\ send\ to\ P4$
3. $t \in \{1..11\}@\ \{P1..P3\}$:
   $(series\ of\ remote\ invocations\ initiated\ in\ P1\ that$
   $result\ in\ reference\ to\ J_{P2}\ being\ exported\ to\ P3;$
   $A_{P1}\ becomes\ unreachable\ locally\ in\ P1;$
   $The\ reference\ from\ the\ GC\ local-root\ to\ K_{P3}\ now\ holds\ the\ entire\ cycle\ globally\ reachable).$
4. $11 \prec t \prec iii@P1$:
   $Take\ snapshot,\ perform\ graph\ summarization,\ now\ includes:$
   $Stub(F_{P2})_{P1} \Rightarrow \{ScionsTo \equiv \{D_{P1}\}, Local.Reach \equiv\ false, IC \equiv\ x+1\}).$
5. $t =\ iii@P4 : Alg_{3a} \Rightarrow\ \{\{\{F_{P2}, x\}, V_{P5}, T_{P4}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}\}\},\ and\ send\ to\ P1$
6. $t =\ iv@P1 : Alg_{4a} \Rightarrow\ \{\{\{F_{P2}, x\}, V_{P5}, T_{P4}, D_{P1}\} \rightarrow \{V_{P5}, T_{P4}, D_{P1}, \{F_{P2}, x+1\}\}\},\ and\ send\ to\ P2$
7. $t \succ\ iv@P2 :\ Matching(Alg_{4a}) \Rightarrow \{\{\{F_{P2}, x\}\} \rightarrow\ \{\{F_{P2}, x+1\}\}\}$

8. $Cycle\ Found \Rightarrow\ false,\ different\ IC\ values\ (x\ and\ x+1)\ for\ F_{P2},\ detection\ abort$

This use of invocation-counters also holds the following advantage: detections already in course for **real** cycles are never aborted due to updates in summarized graph information contained in snapshots; in other words, a detection in course, regardless of when it was initiated can only be aborted if one of its subgraphs was actually touched by the mutator, after it has begun. Thus, there are very loose synchronization requirements for cycle detection; it can be performed lazily without disruption to applications.[4]

Race condition detection in the previous example can be optimized if $P1$ analyzes unmatched invocation-counters, in the algebra it is about to send to $P2$. However, that is not required to maintain safety.

In summary, the safety rules enforced are: i) CDMs sent to non-existent scions are discarded and detection terminated and, ii) different invocation-counter values, in source and target sets of a CDM, for the same object, cause detection to abort.

III.2.2.3 Analysis of Algorithm Properties

In this subsection, we address the relevant properties of a complete distributed garbage collector discussing them against the algorithm proposed: safety, liveness, completeness, termination, and scalability.

Safety: The DCD does not reclaim objects reachable to the mutator because each CDM is only forwarded along a stub if it is not reachable locally. Furthermore, if there are distributed invocations along the hypothetical cycle being traced, these races are detected checking invocation-counters.

The DCD is resilient to CDM loss, delay, re-ordering and replay. CDM loss does not prevent safety. If CDMs never arrive, no distributed cycles can be detected and, therefore, no action is taken by the DCD producing no error. It could, obviously, delay cycle detection.

Similarly, message delay influences cycle detection in two aspects: i) for a real cycle, the later CDMs arrive at, and are forwarded from processes, the later the cycles will be detected; ii) for a "false" cycle, when CDMs are delayed, subsequent mutator events on the hypothetical cycle being detected will abort cycle detection before it reaches again the initiating site.

The algorithm is safe w.r.t message replay because: i) CDM replay will either produce the same negative results (on a "false" cycle), ii) or will abort cycle detection if the cycle has already been reclaimed. Note that there is no state in processes regarding ongoing detections that could become corrupted with later CDM replay or reordering.

There are no ordering requirements, and therefore no competition, nor racing conditions, among CDMs. Simultaneous detections of the same cyclic garbage graph can occur without

[4]Invocation-counters perform a similar function to virtual memory traps in concurrent LGCs (used to find out if an object has been invoked/accessed after it was marked by the LGC(Wilson 1992)).

error. This is due to the fact that snapshots in processes need not hold information about sent or received CDMs.

Liveness: Algorithm liveness naturally depends on CDMs being forwarded by DCD and processes updating their snapshots. When CDMs are queued, they are only so for a period of time, until they are combined with, or suppressed due to, other CDMs. If this does not happen after that period of time, the queued CDMs are forwarded. In a worst case scenario, this may delay cycle detection, but does not preclude liveness.

Completeness: The algorithm achieves completeness in the sense that any cyclic distributed garbage is eventually detected and reclaimed. Distributed garbage cycles comprise objects that eventually will be described in the snapshots taken independently by processes. Since they are garbage and, therefore, never invoked again, their invocation-counters will stop. Eventually, all the right candidates will be selected. Therefore, a CDM being forwarded along the processes comprising each of the cycles, will eventually match every dependency and terminate, regardless of concurrent mutator activity or snapshot update at any process.

Floating-garbage consists of just recently created distributed cycles that cannot be detected until compressed snapshots, at processes, correctly reflect it. The algorithm is conservative in these situations. Obviously, this is an inevitable phenomenon to GC in general. However, the relevant issue in cycle detection is actually and effectively detecting them, since they are stable (and therefore also long-lived).

CDM queueing and group awareness do not preclude completeness since CDMs are eventually forwarded or combined. CDM optimizations do not hinder completeness since the CDM resulting from it, carries the same information, w.r.t. cycle detection, as the CDMs that were combined, or suppressed.

The completeness, and efficiency of any suspect-based approach to cycle detection, ultimately depends on the selection of candidates to be tested. Efficient selection of cycle candidates is an issue on its own. Heuristics found in the literature (e.g., in (Maheshwari and Liskov 1995; Maheshwari and Liskov 1997c; Rodrigues and Jones 1996)) may be used: local un-reachability, distance from GC local-roots, recency of last invocation. For instance, invocation-counters could be used to select, as candidates, objects not invoked for a long time among those that are not reachable locally. Additionally, when a candidate is selected, it should be selected again only after all other possible candidates have been.

Termination: Regarding termination, ongoing cycle detections terminate when: i) a distributed cycle is found, ii) a stub is found to be locally reachable (detection is terminated along that path), iii) a race with the mutator is detected (detection is aborted), and iv) the incoming and out-going CDMs are identical, i.e., no additional information is provided downstream for cycle detection (thus, subsequent CDM forwarding along those stubs would be useless and therefore terminated).

CDMs include saturation-bits (one for each element in the target-set) that are used in CDM comparison. This way, the same message is never forwarded, with unchanged source and target sets, twice through the same path. This also allows detections, traveling through inner cycles, to propagate along these cycles, accounting extra dependencies and target objects. Once information about the whole inner cycle is collected, it can be forwarded outside the cycle through every possible direction. This portrays a two-phase process. The first phase accounts for dependencies and the second ensures their propagation.

The previous rules prevent CDMs from becoming ever-lasting and building up on each other. CDM optimizations, such as CDM queueing for ulterior combination or suppression, further prevent these situations. Termination does not require DCD in each process to maintain state about all ongoing detections that span the process.

Scalability: The scalability of the algorithm stems mainly from the loose synchronization requirements it imposes, since cycle detection is performed lazily, without interfering with DGC. Ongoing cycle detections do not require storing additional state information at participating processes (they are even ignorant of each other w.r.t. this aspect) regarding each one.

Cycle detection does not require causality information w.r.t distributed invocations involving processes, except for the invocation-counters in stub and scions. These can be updated without imposing extra delay to applications (a single counter increment is masked by remote invocation delay).

The algorithm does not require any process to send messages to processes other than those it would normally communicate with, due to remote invocations and acyclic DGC. This is not the case with other approaches based on: backtracking, optimistic backtracking, group creation, or central detection.

CDMs can be piggy-backed in regular communication among processes (remote invocations, acyclic DGC messages). If used, CDM optimizations further reduce the number of CDMs sent, improving scalability. CDM queueing and optimization are performed only as a best-effort approach. Thus, they do not constitute a requirement to maintain state about detections. They may be forwarded immediately, if necessary.

III.2.3 DGC-Consistent Cuts for Replicated Memory

This section describes **D**GC-**C**onsistent **C**uts for **R**eplicated **M**emory (Veiga and Ferreira 2005b)(**DCC-RM**), used to detect and reclaim distributed cycles in replicated memory, i.e., cycles of garbage comprising objects replicated in several processes. This approach extends the notion of DGC-Consistent Cuts (Veiga and Ferreira 2003a) to distributed systems with data replication, thus serving as a complement to an acyclic DGC for replicated objects (Sanchez et al. 2001). It is the first viable solution to complete DGC for replicated memory (RM) systems.

The memory management of distributed (and possibly persistent) graphs of replicated objects is a very difficult task. These graphs are large, widely distributed and frequently modified through assignment operations executed by applications, in local replicas. In addition, data replicated in many processes is not necessarily coherent making manual memory management much harder.

The presence of replication further exacerbates the issue of completeness concerning the automatic memory management of replicated objects. As it will become clear in this section, it only takes one replica of an object, being involved in a distributed cyclic garbage, to consequently encompass all other replicas of the same object in the cycle, preventing their reclamation, and of all other objects referenced by them. Thus, distributed cyclic garbage involving replicated objects is, arguably, more frequent and wastes more storage, when compared with systems without replication (Wilson 1996; Louboutin and Cahill 1997; Richer and Shapiro 2000).

The algorithm proposed employs a centralized approach, in which there is a dedicated server, the distributed cycles detector (**DCD**), in charge of creating replication-aware DGC-Consistent Cuts (i.e., DCCs-RM), by combining compressed snapshots received from application processes. It detects cycles comprised within them, by performing a conservative mark-and-sweep (CMS). It does so asynchronously, without requiring any distributed synchronization among the processes comprising the cycles, and without delaying acyclic DGC. The algorithm takes replication into account (by enforcing the Union Rule) while creating the cuts, and performing CMS on them.

The DCC-RM approach cumulatively addresses the two fundamental problems concerning DGC in RM systems: i) ensuring safety in presence of replication, and ii) achieving completeness. Previous proposals found in the literature are either:

- not safe w.r.t. replication (see Section III.1.2).
- expressly stated as incomplete (Sanchez et al. 2001).
- only able to detect cycles that do not span processes, e.g., (Ferreira and Shapiro 1994a; Ferreira and Shapiro 1996; Blondel et al. 1998), which is also incomplete.
- impose full-stop in all processes for sequential distributed tracing, e.g., (Le Sergent and Berthomieu 1992; Kordale et al. 1993; Yu and Cox 1996), which is clearly not viable, more so in wide area systems.

The algorithm achieves both safety and completeness by obeying the Union Rule (already brought up in Section III.1.4) while performing CMS on the DCC-RM. Snapshot compression,

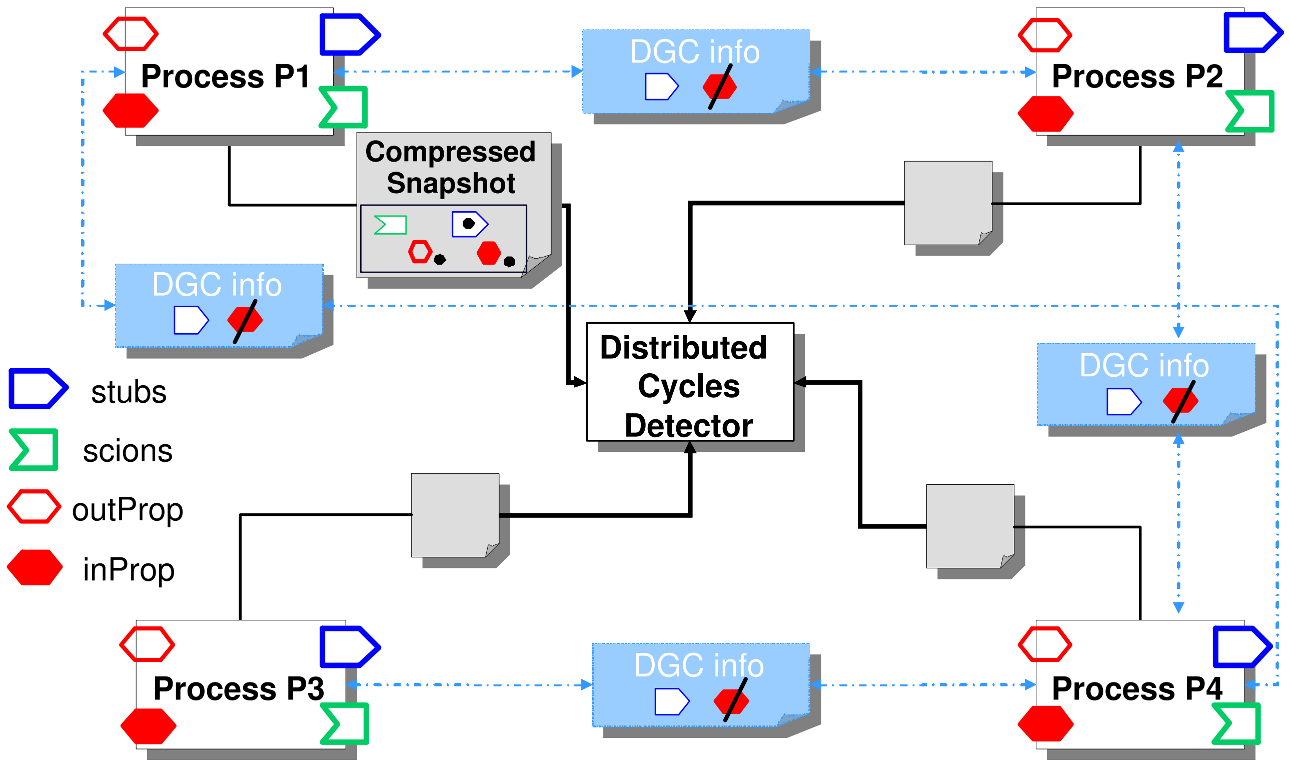


Figure III.2.14: Algorithm overview

performed seldom by processes, is also done obeying the Union Rule, thus capturing the implicit associations among the various replicas of the same objects.

III.2.3.1 Overview

Figure III.2.14 lays out an overview of distributed garbage collection in a RM system. It depicts four processes in an example situation analogous to the ones presented in the previous sections. In this case, each process maintains references to objects located in other processes, and replicates them to be invoked locally.

There is an acyclic DGC component running in each process, that is integrated with replication management, that handles DGC and object replication structures. It sends messages to other processes to inform them of its DGC structures (e.g., *NewSetStubs* message, and *Unreachable* messages to handle GC of replicas), and to instruct them to perform modifications on their own DGC structures (e.g., *Reclaim* messages).

As the acyclic DGC is not complete, there is a dedicated distributed cycle detector (DCD) in operation to achieve completeness. From time to time, each process sends a compressed snapshot to the DCD that includes extended information about DGC and replication structures. As the DCD receives the snapshots, it creates DCCs-RM.

The DCD performs CMS on each DCC-RM created, to detect cyclic garbage, possibly involving replicated objects, that is completely comprised in the cut. The DCD drops older snapshots from processes when they are no longer referenced from any DCC-RM. The DCD breaks distributed cycles by issuing special messages, regarding DGC and replication structures.

The rest of this section is organized as follows. The next subsection characterizes the RM model assumed by the DGC algorithm; it is rather abstract and general, so that the GC solutions provided are widely applicable. In the remaining subsections we: i) briefly describe the replication-aware acyclic DGC algorithm used, ii) present the algorithm for cycles detection and reclamation, iii) introduce a prototypical example that is not handled by previous solutions in the literature, and iv) discuss the algorithm properties.

III.2.3.2 Replicated Memory Model

We now describe a model for replicated memory that defines the environment for which the DGC algorithms are conceived. It imposes minimal requirements; basically that these systems support object replication. The model is rather general and can be mapped to existing architectures and systems, both in local (e.g., DSM), or wide area networks (e.g., PerDis). It is used by OBIWAN in its Replication Management and Memory Management modules.

A RM system is a replicated distributed memory spanning several processes. Instead of performing remote invocations, processes access data always locally. In other words, application code inside a process never sends messages explicitly. Transparently to the application code, the RM runtime system is responsible to replicate data locally when needed.

Each participating process in the RM system encloses the following entities: memory, mutator, and a coherence engine. In our RM model, for each one of these entities, we consider only the operations that are relevant for GC purposes.

III.2.3.2.1 Memory Organization

An **object** is defined to be a consecutive sequence of bytes in memory. Applications can have different views of objects and can see them as language-level class instances, memory pages, data base records, web pages, etc.

The unit for replication is the object. Any object can be replicated in any process. A replica of object X in process P is noted X_P. Each process can hold a replica of any object for reading or writing according to the coherence protocol being used. This does not preclude the possibility of replicating several objects in a single operation; it simply does not impose it.

III.2.3.2.2 Mutator model

The single operation executed by mutators, which is relevant for GC purposes, is **reference assignment**; this is the only way for applications to modify the graph of objects.

The reference assignment operation executed by a mutator in some process P is noted $X := Y_P$. This means that a reference contained in object X is assigned to the value of a reference contained in object Y.[5] If Y points to an object Z in some other process, this assignment operation results in the creation of a new inter-process reference from X to Z.

Obviously, other assignments can delete references transforming objects into garbage. For example, in process P the mutator may perform $(X := NULL)_P$; this may result on some object Z becoming globally unreachable, i.e. garbage, given that there are no references pointing to it. In conclusion, assignment operations (done by mutators) modify the object graph either creating or deleting references.

III.2.3.2.3 Coherence Model

The coherence engine is the entity of the RM system that is responsible to manage the coherence of replicas. The coherence protocol effectively used varies from system to system and depends on several factors such as the number of replicas, distances between processes, and others.

Ideally, the DGC should be orthogonal w.r.t. whatever mechanism is used to maintain replicas coherent. Thus, the only coherence operation, which is considered relevant for DGC purposes, is the **propagation** of an object, i.e. the replication of an object from one process to another. The propagation of an object Y from process $P1$ to process $P2$ is noted $propagateY_{P1 \rightarrow P2}$.

The propagation of an object is performed via an implicit *propagate* message that carries the actual object content. When an object is propagated to a process, its enclosed references are **exported** from the sending process to the receiving process. On the receiving process, i.e. the one receiving the propagated object, the object's enclosed references are **imported**.

We assume that any process can propagate a replica from another process into itself, when the mutator causing the propagation holds a reference to the object being propagated. Thus, if an object X is unreachable locally in process $P1$, the mutator in that process can not force the propagation of X to some other process; however, if some other process $P2$ holds a reference to X, it can request X to be propagated from $P1$ to $P2$.

In a RM system mutators may create inter-process references very easily and frequently, through a simple reference assignment operation (see Section III.2.3.2.2). Note that when such an assignment does result in the creation of an inter-process reference, this can only happen because, in the local process, there was already an object replica containing that reference to the remote object. Thus, inter-process references are created as a result of the propagation of replicas. Such propagation leads to the export and import of references.

In each process, the coherence engine must hold two data structures, called **inPropList** and **outPropList**. Usually, this information already exists in the coherence engine in order to manage the replicas (e.g., they are included in the Replication Management module, in OBIWAN, and have already been presented in Section I.2.2). Thus, they are not a special requirement imposed by DGC, though their existence is leveraged by it.

[5] This notation is not fully accurate but it simplifies the explanation of the DGC algorithm. As a matter of fact, to be more precise we should write $X.ref = Y.ref$ (C++/C#/Java style notation). However, this improved precision is not important for the DGC algorithm description and would complicate it un-necessarily.

Each entry in these lists indicates the process *from which* each object has been propagated, and the processes *to which* each object has been propagated, respectively. Thus, each entry of the **inPropList**/**outPropList** contains the following information:

- **propObj** - the reference of the object that has been propagated into/to a process;
- **propProc** - the process from/to which the object *p*ropObj has been propagated;
- **sentUmess**/**recUmess** - bit indicating if a $Unreachable$ message has been sent or received. $Unreachable$ messages refer to un-reachability from GC local-roots. They have already been described in Section III.1.4.4 and will be addressed, in the context of this algorithm, later in Section III.2.3.3.3.

For clarity and brevity, entries in the **inPropList** and **outPropList** will be hereafter referred to as *inProp*/*outProp* entries, or simply as *inProps*/*outProps*.

It's worthy to note that in the RM model, the only way a process can create inter-process references is through the execution of two operations: (i) **reference assignment**, which is performed explicitly by the mutator and (ii) **object propagation**, which is performed by the coherence engine in order to allow the mutator to access some object.

As an example, in some DSM-based systems, when the mutator tries to access an object that is not yet cached locally, a page fault is generated; then, this fault is automatically recovered by the coherence engine that obtains a replica of the faulted object from some other process. In OBIWAN, this is performed by the incremental replication mechanism, triggered by the invocation of ProxyOut objects.

III.2.3.3 Acyclic Distributed Garbage Collection

The acyclic DGC for the RM model is based on (Sanchez et al. 2001), with some extensions and optimizations that are described in the remainder of the section. It is assumed that a tracing-based LGC is running in each process, as it is common usage. LGC will be further mentioned when describing interaction with DGC.

III.2.3.3.1 Replication Awareness

In Section III.1.4 we presented algorithms that are safe in the presence of replication, as opposed to those in Sections III.1.2 and III.1.3 that are not. We now exemplify how a replication-aware DGC operates.

Consider Figure III.2.15 in which an object X is replicated in processes $P1$ and $P2$. Now, suppose that X_{P1} contains a reference to an object Z in another process $P3$, X_{P1} points to no other object, X_{P1} is not reachable locally (i.e., to the mutator in $P1$), while X_{P2} is reachable locally in $P2$.

Then, the question is: should Z_{P3} be considered garbage? Classical DGC algorithms (designed for function-shipping systems) consider that Z_{P3} is effectively garbage. However, this is

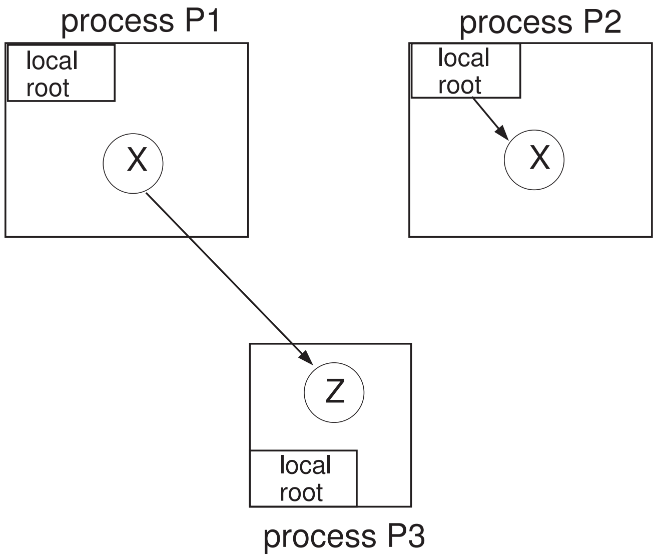


Figure III.2.15: *Safety problem of DGC algorithms which do not handle replicated data: Z is erroneously considered unreachable.*

wrong because, in a RM system, it is possible for an application in $P2$ to acquire a replica of X from some other process, in particular, X_{P1}. Thus, the fact that X_{P1} is not reachable locally in process $P1$ does not mean that X is unreachable globally. As a matter of fact, according to the coherence model, X_{P1} contents can be accessed by an application in process $P2$ by means of a propagate operation.

Therefore, in a RM system, a target object Z is considered unreachable only if the union of all the replicas of the source object, X in this example, do not refer to it.

III.2.3.3.2 Data Structures

The acyclic DGC collector manages a combined set of data structures containing typical DGC structures, as well as data structures relative to replication management (defined by the coherence model). Thus, it manages: i) **scions**, ii) **stubs**, iii) **outProps**, and iv) **inProps**.

Each data structure is extended to hold an additional **time-stamp** value, provided by a monotonic counter global to the enclosing process. These time-stamp fields are mainly required for cycle detection (i.e., for constructing DCCs-RM) but they also serve additional purposes, explained afterwards. The counter is shared among scions and outProps, in order to maintain a total order, within each process, among events relevant to DGC. W.r.t. to stubs and scions, it has already been described how these time-stamps prevent race conditions w.r.t. NewSetStubs messages (see Section III.2.1.1.1). In the case of outProps and inProps, time-stamps are also useful to ensure safety, while optimizing storage, as further described in III.2.3.3.3.1.

III.2.3.3.3 Messages

The acyclic DGC components running on each process exchange messages of the following types:

- **NewSetStubs**: This message is distinctive of algorithms based on reference-listing, as the case of this one. It has already been described in Section III.1.2.1.5, and further addressed in Section III.2.1.1.1. It carries the identification of stubs, in the sender process, concerning inter-process references targeting objects in the receiving process.
- **Unreachable(**$obj,\ time - stamp$**)**: this message is sent to the process where an object was originally replicated from when, in the sender process, it is reachable only from the `inPropList`. This sending event is registered by changing a `sentUmess` bit in the corresponding `inPropList` entry from 0 to 1. When a valid $Unreachable$ message is delivered to a process, this delivery event is registered by changing a `recUmess` bit in the corresponding `outPropList` entry from 0 to 1. Validity of Unreachable messages is addressed in the next paragraph III.2.3.3.3.1.
- **Reclaim(**$obj,\ time - stamp$**)**: this message is sent to all processes that have previously replicated an object from the sender process. It is sent when the object is reachable only from the `outPropList`, and the sender process has already received valid $Unreachable$ messages from all the processes to which that object has been previously propagated. After the messages are sent and acknowledged, the corresponding entries in the `outPropList` are deleted; otherwise, nothing is done. When a process receives a valid $Reclaim$ message it deletes the corresponding entry in its `inPropList` (validity of Reclaim messages is addressed next in paragraph III.2.3.3.3.1).

In summary, besides the message $NewSetStubs$, two other messages may be sent by the acyclic DGC: $Unreachable$ and $Reclaim$. On the receiving process, these messages are handled by the acyclic DGC that performs the following operations: sets the `recUmess` bit in the corresponding `outPropList` entry, and deletes the corresponding entry in the `inPropList`, respectively. Thus, a replicated object is effectively reclaimed (by the LGC) only after the corresponding entry in the `inPropList` is deleted.

III.2.3.3.3.1 *Unreachable/Reclaim* message validity and In/OutPropList optimization

In the original approach proposed in (Sanchez et al. 2001), each time an object (e.g., X) is propagated (i.e., replicated) between two processes (e.g., from $P2$ to $P1$), the corresponding outProp is created in $P2$, and the corresponding inProp is created in $P1$. This takes place even if the object has already been propagated before from $P2$ to $P1$. This is required to ensure safety, and avoid race conditions, in the following situations.

Some time after the first propagation from $P2$ to $P1$, object replica X_{P1} may have become unreachable locally in $P1$. The acyclic DGC in $P1$ notifies its counterpart in $P2$ of this event, lazily, by sending a *Unreachable* message w.r.t X_{P1} some time afterwards. Later on, the mutator in $P1$, transversing other references in its graph, may come across a reference to X_{P2} and, possibly, refresh its replica X_{P1}, thus re-propagating X_{P2} to $P1$. If $P1$ intends to re-use the existing replica, $P2$ must still be informed and will reply with a propagate message, in this case with *empty* object content.

Due to the asynchronous nature of the DGC w.r.t. the mutator, it may happen that the *Unreachable* message is actually sent to $P2$ only after the second propagation occurs. This may lead $P2$ to believe that X_{P1} is no longer reachable locally in $P1$, and if by that time, X_{P2} is also unreachable locally in $P2$, both replicas may be reclaimed. This is clearly incorrect since X_{P1} is being used by the mutator in $P1$. In the original proposal, this is handled by keeping multiple inProps and outProps regarding the same object, one for each propagation, which ensures safety in these situations, since the second pair of inProp/outProp entries would prevent the replicas from being reclaimed. Thus, all Unreachable and Reclaim messages were considered valid upon delivery.

The extension of inProps and outProps with an additional time-stamp field, besides helping in the construction of each DCC-RM, allows the following optimization. Each time an object is re-propagated between two processes (e.g., the same $P1$ and $P2$), the existing inProp and outProp entries are re-used. The *sentUMess* bit of the outProp in $P2$ is reset, and the time-stamp field is refreshed. This value is included in the *propagate* message, sent from $P2$ to $P1$ and used to refresh the time-stamp field of the corresponding inProp in $P1$.

This poses the question of validity of *Unreachable* and *Reclaim* messages. These messages must contain the time-stamp of the moment when the process decides to send them. For the message to be valid, this time-stamp must be current, i.e., be related to the last time the object was propagated. Thus, in the previous example, messages with older time-stamps are simply ignored, i.e., they are deemed as invalid. This ensures safety, while avoid the need for storing individual inProp or outProp entries each time an object is propagated, between the same pair of processes.

Similarly, each *Reclaim* message also carries the time-stamp value of the outProp entry when the process decided to send it. This ensures safety when receiving delayed *Reclaim* messages.

III.2.3.3.4 Safety Rules

Whatever the coherence protocol, there is only one interaction of the mutator with the acyclic DGC algorithm. This interaction is twofold: i) immediately before a propagate message is sent, the references being exported (contained in the propagated object) must be found in order to create the corresponding scions, and ii) immediately before a propagate message is delivered, the outgoing inter-process references being imported must be found in order to create the corresponding local stubs, if they do not exist yet. Note that this may result in the creation of chains of stub-scion pairs, as it happens in the SSP Chains algorithm (Shapiro et al. 1992b).

To summarize, the following rules are enforced by the acyclic DGC:

- **Clean Before Send Propagate**: Before sending a propagate message, enclosing an object Y, from a process $P2$, Y must be scanned for references and the corresponding scions created in $P2$.

- **Clean Before Deliver Propagate:** Before delivering a propagate message, enclosing an object Y, to a process $P1$, Y must be scanned for outgoing inter-process references and the corresponding stubs created in $P1$, if they do not exist yet.

From these rules, results the fact that scions are always created before the corresponding stubs; and outProps are always created before their corresponding inProps. This is due to a causality relationship (their creation is causally ordered) between them.

The algorithm also enforces the Union Rule, stated as follows:

- **Union Rule**: a target object Z is considered unreachable only if the union of all the replicas of the source objects do not refer to it.

This rule is enforced by how the algorithm manages In/OutPropLists and handles Unreachable/Reclaim messages, as already described in Section III.2.3.3.3. A replica is considered unreachable only when all other replicas of the same object are also unreachable.

III.2.3.4 Algorithm for Cyclic Distributed Garbage Collection

This section describes the algorithm to detect cyclic garbage comprising replicated objects, spanning across several processes. It describes the creation and handling of DGC-Consistent Cuts for Replicated Memory (Veiga and Ferreira 2005b), that complements the acyclic DGC just presented. Given that this approach is based on DGC-Consistent Cuts, already presented in Section III.2.1, we highlight mainly those aspects related to replication-awareness.

III.2.3.4.1 Data Structures

The DCD requires that the data structures maintained by the acyclic DGC, i.e., **stubs**, **scions**, **inProps** and **outProps**, be extended with time-stamps, as described before in III.2.3.3.2. Additionally, it defines the following data structures:

- **Vector-Clock**: each process maintains a record of the highest time-stamp associated to the creation of scions and outProps, known from all processes constituting, including itself, in a vector-clock (Mattern 1989).
- **Snapshot**: representation of the object graph of a process, including its stubs, scions, outProps, inProps, and the vector-clock maintained by the process. Snapshots are subject to compression described in III.2.3.4.4.
- **DCC-RM**: a conservative juxtaposition of snapshots taken at uncoordinated times, comprised of, at most, one snapshot concerning each process, that is extended to be replication-aware, thus conforming to the Union Rule.

The acyclic DGC component in application processes creates and maintains stubs, scions, inProps, and outProps. It also manages vector-clocks. The cyclic DGC component in each process generates snapshots. DCCs-RM are created and used exclusively in the context of the processes assigned to DCD. Besides DCCs-RM, the DCD has access only to snapshots received from application processes. It never manipulates acyclic DGC structures directly.

III.2.3.4.2 Messages

The algorithm defines three types of messages: **NewSnapshot**, **DeleteScion**, and **DeclareUnreachable**. The first one is descriptive and the last two are operative. NewSnapshot and DeleteScion messages are similar to what was described in Section III.2.1.1.2, with the exception that the snapshots included in NewSnapshot messages also carry information regarding object replication. The messages of type DeclareUnreachable are defined as follows.

- **DeclareUnreachable**: message sent from a DCD to the cyclic DGC component of an application process, instructing it to explicitly declare a specific inProp entry as regarding an object that belongs to cyclic garbage. This equates to considering the object as neither reachable from GC local-roots in the process, nor via scions, from other processes.

  From this moment on, the acyclic DGC is able to make progress towards reclaiming the distributed cycle. The process receiving the DeclareUnreachable message, after the next LGC, will send a Unreachable message to the process holding the counterpart outProp entry and this fact will be registered in both sentUMess and recUMess bits.

The approach employed to break links among objects comprised in cyclic garbage differs w.r.t. scions and inProps/outProps. This stems from the following differences in the way these structures are handled by the acyclic DGC.

W.r.t. scions belonging to cyclic garbage, they can be simply eliminated, by instruction of the DCD, in the knowledge that their corresponding stubs do not protect objects from being reclaimed by the LGC. Stubs are only created/preserved while objects are still reachable globally, which is precisely what is annulled by deleting scions, thus breaking the distributed cycle. As a result, the corresponding stubs will eventually cease to exist after future executions of the LGC.

W.r.t. inProps, as opposed to stubs, they are preserved until explicitly deleted via a valid Reclaim message. Thus, the DCD cannot simply instruct deletion of outProps as this would create inProp ”zombies”, that is, inProp entries, about which, no Reclaim message would ever be received in the future. This would obviously prevent algorithm progression, thus hindering liveness and completeness.

One other flawed alternative would be for the DCD to send instructions setting the recUMess bit, in all outProps regarding an object found to be garbage. That, although saving message traffic, would violate the acyclic DGC rules. It would imply sending Reclaim messages regarding outProps, for which Unreachable messages were never received. Furthermore, it would imply deleting inProp entries that do not have the sentUMess bit set.

III.2.3.4.3 Cycle Detection

Cycle detection is performed by the DCD. It is divided in four distinct phases: i) snapshot reception, ii) DCC-RM creation, iii) conservative mark-and-sweep (CMS), and iv) sending of messages for cycle dismantlement. These phases are analogous to the four phases presented in DGC-Consistent Cuts (see Section III.2.1.1.3).

Snapshot Reception: This phase is performed in the same manner as has already been described in Section III.2.1.1.3. This phase can be carried out concurrently with any other. The DCD is always ready to receive snapshots from processes, possibly keeping several versions from the same process. Once no longer involved in ulterior phases, older snapshots from each process are discarded.

DCC-RM Creation: The DCD creates a DCC-RM by assembling one snapshot from a number of processes, not necessarily from all that are available. The DCD is only able to detect cycles fully comprised in a DCC-RM. The same aspects regarding size, scope, asynchronism, uncoordination and liveness, described in Section III.2.1.1.3 w.r.t. DGC-Consistent Cuts, also apply, by construction, to DCC-RM creation. Once a snapshot is included in a DCC-RM, it is neither modified nor replaced in the DCC-RM. When a DCC-RM is created it can be forwarded to phase CMS (phase iii), and stored for later combination with other DCCs-RM or to be sent to other DCDs.

In Figure III.2.16, we show in bold, on the left-hand side a cut which is not consistent for causality purposes. It depicts a situation that could never have happened during the operation of the acyclic DGC. It results from the reception of snapshots from different processes, taken and sent to the DCD without coordination. Nevertheless, the DCD is able operate safely w.r.t. DGC, during CMS.

Object graphs received by the DCD provide a view of the global graph that does not correspond to a real one. The differences, i.e., the GC and replication structures that should also be included in the DCC-RM, but are not represented in individual snapshots received by the DCD, are shaded in the figure. Since they can occur at any time, executions of LGC are not represented.

The global graph as perceived by the DCD, is represented on the right-hand side of Figure III.2.16. The DCC-RM presented is thus a set of GC and replication structures that allows a safe view, w.r.t. cyclic DGC, of the distributed object graph. The safety of this view stems from the rules that define the root-set of the CMS (the next phase to be described).

There is a specialized heuristic in use when constructing a DCC-RM. When a snapshot from a process is included in the DCC-RM, the DCD checks the availability of snapshots from other processes involved in replication operations, i.e., processes that have replicated objects from/to the process already included in the DCC-RM. The DCD adds to the DCD-RM the snapshots of one or a few, randomly chosen among such processes, that are available at the DCD. This can be performed efficiently because GC and replication structures, inside the snapshots, are grouped by corresponding process.

The heuristic aims at the creation of a DCC-RM encompassing snapshots from processes holding replicas of the same objects. This is required because a distributed cycle comprising replicated objects can only be reclaimed if all replicas are found to be unreachable. When there are required snapshots unavailable, or the size of the DCC-RM reaches a specified limit, the DCD may store it for ulterior combination (more details in Section III.2.3.4.4)

Conservative Mark-and-Sweep: After the DCC-RM is constructed, it is subject to a replication-aware CMS, that is performed within the DCD process. It ensures safety, taking causality into account, w.r.t. GC and replication structures. This guarantees that the DCC-RM is consistent for cycle detection. There is no exchange of messages with other processes, during this phase. The marking is performed on a separate bit-map, as described before for CMS of DGC-consistent cuts (v. Section III.2.1.1.3).

The extended root-set used in CMS is comprised of:

1. Those objects that, in each application process, are directly reachable from the GC local-roots (stack, etc.) must be obviously considered roots of the CMS.

2. Scions whose corresponding stubs are included in processes whose snapshot is not included in the DCC-RM, are also members of the CMS root-set, for safety reasons. As a matter of fact, such scions may not have a corresponding stub (so they could be simply discarded) but the DCD can't say that for sure. Thus, it uses a conservative approach.

3. inProp and outProp entries, whose corresponding outProps and inProps belong to processes not included in the DCC-RM, must also be considered as members of the CMS root-set. This is a conservative approach, once again, to ensure safety.

4. Scions with time-stamp greater than the highest time-stamp (regarding the process where the targeted object resides), known by the process holding the corresponding stub. This is also a conservative approach. These scions are those whose corresponding stubs have not yet been created when the referring process recorded its snapshot. These scions verify the following condition:

   $scion.timestamp > VC_{P_{stub}}[P_{scion}]$

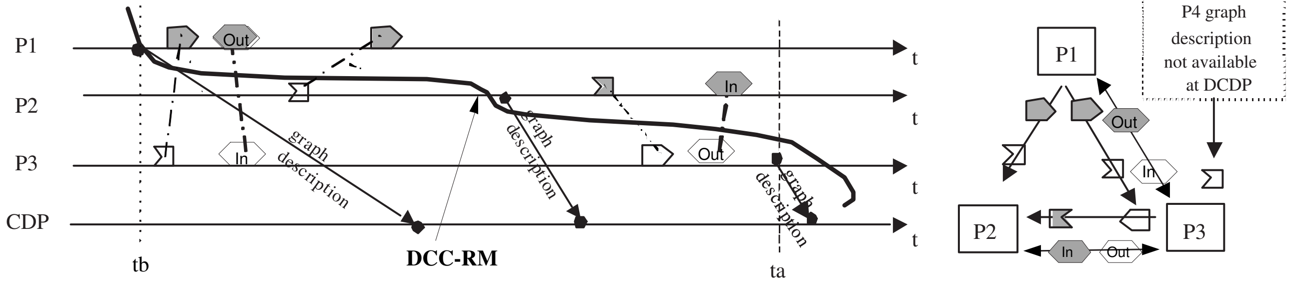


Figure III.2.16: DCC-RM and process snapshots as seen by the DCD.

Recall that $scion.timestamp$ is the time-stamp given to the scion when it was created, $VC_{P_{stub}}$ is the vector-clock maintained by the process holding the corresponding stub, and P_{scion} is the identifier of the process holding the scion.

5. outProp entries, with time-stamp greater than the greatest value known in the process holding the corresponding inProp entry (w.r.t. where the outProp resides). This is because the snapshot of the process holding the inProp is not current enough. They are defined by the following condition:

   $outProp.timestamp > VC_{P_{inProp}}[P_{outProp}]$

   where $outProp.timestamp$ is the time-stamp given to the outProp when it was created, $VC_{P_{inProp}}$ is the vector-clock kept at the process holding the corresponding inProp, and $P_{outProp}$ is the identifier of the process holding the outProp.

6. inProp entries, with time-stamp greater than the greatest value known in the processing holding the corresponding outProp entry (w.r.t. where the inProp resides). This is because the snapshot of the process holding the outProp is not current enough. They are defined by the following condition:

   $inProp.timestamp > VC_{P_{outProp}}[P_{outProp}],$

   where $inProp.timestamp$ is the time-stamp the inProp obtained from corresponding outProp,
   $VC_{P_{outProp}}$ is the vector-clock kept at the process holding the corresponding outProp, and $P_{outProp}$ is the identifier of the process holding the outProp.

   Notice that the asymmetry between the last and previous conditions stems from the fact that inProps get the time-stamps from their corresponding outProps, when the latter are created.

The last three items in the CMS root-set enforce a conservative approach to ensure safety. The scions in this situation are those whose corresponding stubs have not been created yet when their enclosing application process generated its snapshot. Similarly, the outProp entries, included in the root-set, are those whose corresponding inProp entries had not been created, when the snapshot of their enclosing process was taken.

Note that the situations, described in the last three items, may occur because the graph descriptions received by the DCD are snapshots taken at different moments at different processes, with no coordination at all. This is a consequence of the fact that there is no need for global synchronization among participating processes, w.r.t. generating snapshots and sending them to the DCD.

Nonetheless, not all inProp and outProp entries in the DCC-RM are included in the root-set of the CMS; only those required for safety. If that were the case, then no distributed cycle comprising replicated objects could ever be detected.

Starting from the root-set of the DCC-RM, the DCD performs a replication-aware mark-and-sweep, in such a way that inter-process references are traced only if the corresponding stub-scion pair exists in the DCC-RM. Similarly, corresponding outProp and inProp entries, indicating replication paths, are also traced by the CMS, bi-directionally. They are regarded as implicit

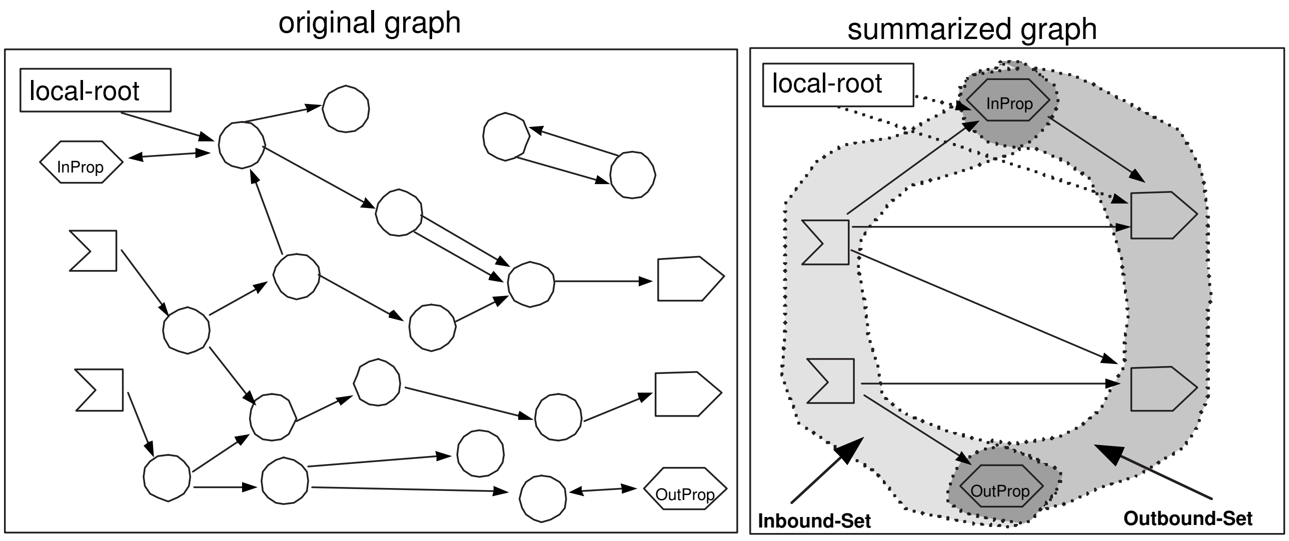


Figure III.2.17: Summarization of an object graph into $InboundSet$ and $OutboundSet$.

inter-process references, in order to uphold the Union Rule. Otherwise, the tracing on those entries stops.

As a result of the CMS, GC and replication structures, belonging to distributed garbage cycles fully comprised within the DCC-RM, are not marked. Next, we describe how detected cycles are dismantled, i.e., broken.

Sending of Messages for Cycle Dismantlement: It is safe to explicitly delete scions, and declare inProp entries as Unreachable, when they are found to be garbage, in order to break the distributed cycles detected. Therefore, w.r.t. some of the unmarked structures, their corresponding processes can be sent DeleteScion and DeclareUnreachable messages. This selection will not hinder completeness, but just affect bandwidth and the speed of reclamation. Messages regarding the same process may be queued and batched. Acknowledged messages are recorded, in a best-effort approach, to prevent duplicate messages if the same cycle is detected in more than one DCC-RM.

III.2.3.4.4 Optimizations

The three optimizations, presented in Section III.2.1.1.4 can also be applied to the algorithm: i) snapshot compression, ii) multiple DCDs, and iii) hierarchical DCDs.

Snapshot Compression: Compression is applied to snapshots so that references strictly internal to process, and scalar object contents, are discarded to save bandwidth, storage and reduce DCD load. As explained before, they are not relevant for DCD purposes. The compressed snapshots contain stubs, scions, inProps and outProps. A compressed snapshot registers these structures and all relevant associations, not only among scions and stubs, taking the Union Rule into account.

Summarization transforms a snapshot of an application graph into two sets: $InboundSet$ and $OutboundSet$. They play the same role as scions and stubs in the context of DGC-Consistent Cuts. Thus, the $InboundSet$, in a snapshot, includes all the entries of the above mentioned data structures that can propagate reachability marks *inside* a process, i.e., scions, and outProp and inProp entries. The reason to include inProp entries along with outProp entries stems from the need to uphold the Union Rule. This way, if a replica is marked as reachable, every other replica of the same object must be so as well. Thus, reachability marks in CMS must be propagated both ways, thus through outProp and inProp entries.

Accordingly, the $OutboundSet$ in a process includes all the entries of the above mentioned data structures that can propagate reachability marks *outside* a process, i.e., stubs, and inProp and outProp entries. The need to include outProp entries along with inProp entries, is symmetrical to the previous case. It also stems from the need to uphold the Union Rule.

Additionally, every entry belonging to the $InboundSet$ must include the set of entries of the $OutboundSet$ that are (transitively) reachable from it. This set is named $OutboundFrom$.

Finally, every entry in the $OutboundSet$ must bear a special bit indicating local reachability, i.e., if there is a transitive path from a GC local-root of the enclosing process, that can lead to this entry.

The notions of $InboundSet$ and $OutboundSet$ are illustrated generally in Figure III.2.17. Note that to uphold the Union Rule, inProp and outProp entries belong to both $InboundSet$ and $OutboundSet$ of the summarized graph. This is made explicit in the original graph by using double-direction arrows between these entries and the objects they refer to. In the summarized version, this is made clear with different shadings: brighter for $InboundSet$ and darker for $OutboundSet$.

Multiple DCDs: As described before, even though the DCD performs cyclic garbage collection in a centralized manner, there is no impediment that multiple DCDs be used for increased availability, performance and scalability.

Hierarchical DCDs: Completeness and scalability of the algorithm, w.r.t. size of distributed cycles, is achieved using an hierarchical approach already discussed for DGC-Consistent Cuts. We discuss now some additional points that are relevant, w.r.t. hierarchical DCDs, when using DCC-RM to detect distributed cycles comprising replicated objects.

The algorithm does not require all objects, belonging to a cycle comprising replicated objects, to be co-located in the machine, as in (Ferreira and Shapiro 1994a) where all bunches containing replicas of an object must be co-located in the same process.

On the contrary, the algorithm's requirement is only that all replicas of the objects, comprised in such a cycle, be encompassed by the snapshots included in a DCC-RM. This raises two significant differences w.r.t. previous work. First, the DCD does not actually hold object content, thus can handle a much larger number of processes. Second, even if the distributed cycle spans a large number of processes, the size of the DCC-RM necessary to detect it may be lowered if it is composed hierarchically based on lower-level DCCs-RM (possibly sent by other DCDs).

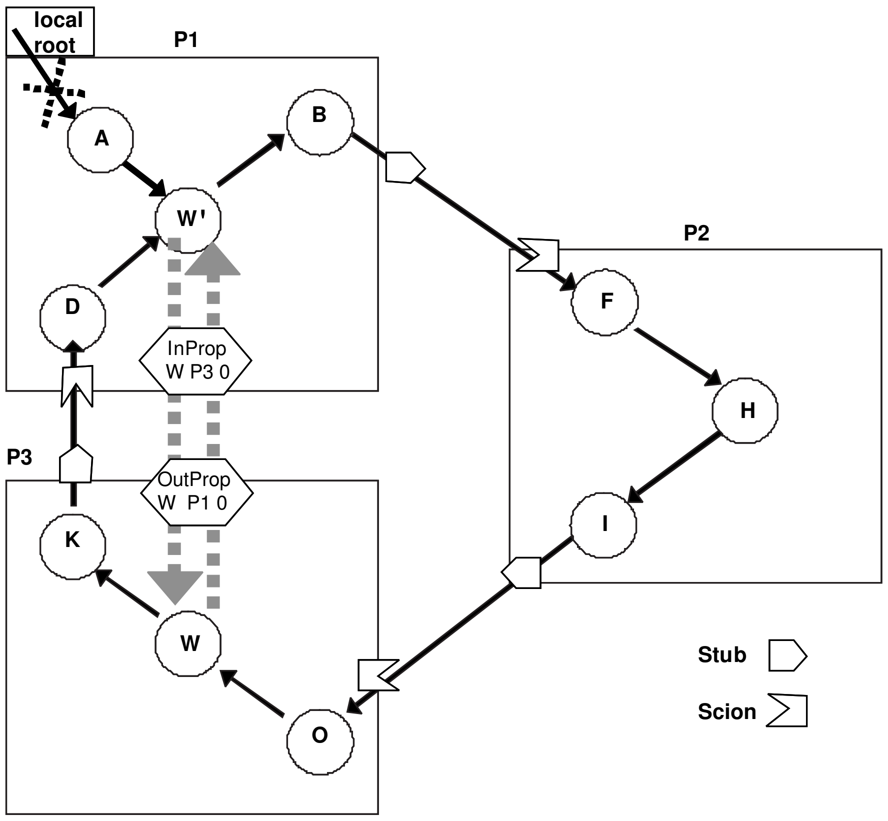


Figure III.2.18: Cyclic distributed garbage comprising replicated objects. Object W has been previously replicated from process $P3$ to process $P1$.

III.2.3.5 Prototypical Example

We now describe how the DCD handles a prototypical example of a distributed cycle comprising replicated objects. The example portrayed is not detectable by previous work, without incurring in full-stop of all the involved processes. We use the same notation and language simplifications (namely with reachability among GC structures, within snapshots) employed for previous examples.

Remote references are described by their corresponding stubs and scions (e.g., $B_{P1} \rightarrow F_{P2}$). Objects replicated from/to processes (e.g., W'_{P1}) are represented with their associated inProp/outProp entries. Furthermore, the association among replicas of the same object, in different processes, is made explicit by gray dashed lines. This eases visualization of the Union Rule presented earlier.

A simple example of a distributed cycle, comprising replicated objects, can be seen in Figure III.2.18. This cycle can be represented by the following (others possible) chain of objects (starting and finishing in $P2$):

$$\{\{F,\ H,\ I\}_{P2}, \{O,\ W,\ K\}_{P3}, \{D,\ W',\ B\}_{P1}\}$$

We chose to depict a scenario where all the snapshots required to detect the cycle are available to the DCD, and are referenced by a DCC-RM. The operation of the algorithm in the absence of information from certain processes has been described in III.2.3.4. It could be portrayed with an example analogous to the one in Figure III.2.5.

Clearly, all objects belong to a distributed garbage cycle, since none of them is reachable from any GC local-root (the one in process $P1$ targeting object A_{P1} has been deleted by the mutator). Therefore, there are no sources of reachability marks. However, in this situation, the acyclic DGC algorithm is unable to proceed, because it conservatively considers objects to be live when they are reachable from scions.

Replicas W_{P3} and W'_{P1} must both be found unreachable for any (or both) of them to be reclaimed. However, both of them are targeted by other objects (O_{P3}, D_{P1} respectively) that are reachable remotely (due to I_{P2}, K_{P3} respectively).

Thus, the acyclic DGC algorithm presented earlier will never issue $Unreachable$ and consequently, $Reclaim$ messages regarding replicas W_{P3} and W'_{P1}. Conversely, without receiving these messages, processes $P1$ and $P3$ will remain including stubs regarding remote references ($B_{P1} \rightarrow F_{P2}$ and $K_{P3} \rightarrow D_{P1}$, respectively) in their $NewSetStubs$ messages. Due to this double inter-dependency, the acyclic DGC algorithm always perceives the objects included in the example portrayed, as reachable globally (therefore, as live objects), while in fact, they are no longer reachable to the mutator.

The summarized graph information at process $P1$ would hold the following data:[6]

- $Inbound - Set(P1) \Rightarrow \{Scion(D_{P1})_{P1},\ InProp(W'_{P1})_{P1}\}$
- $Outbound - Set(P1) \Rightarrow \{Stub(F_{P2})_{P1},\ InProp(W'_{P1})_{P1}\}$
- $Scion(D_{P1})_{P1} \Rightarrow \{OutboundFrom \equiv \{Stub(F_{P2})_{P1},\ InProp(W'_{P1})_{P1}\}\}$, this means that, in $P1$, $Scion(D_{P1})$ leads *to* $Stub(F_{P2})$ and $InProp(W'_{P1})$ (describing replication via W_{P3}).
- $Stub(F_{P2})_{P1} \Rightarrow \{Reach.LOCAL \equiv false\}$, this means that $Stub(F_{P2})$ is not reachable from the local root-set of process $P1$ ($Reach.LOCAL$ is $false$). Note that while the stub refers to an object located in process $P2$, the stub structure is kept *at* process $P1$ where the remote reference actually exists.
- $InProp(W'_{P1})_{P1} \Rightarrow \{OutboundFrom \equiv \{Stub(F_{P2})_{P1},\ InProp(W'_{P1})_{P1}\},$
  $Reach.LOCAL \equiv\ false\}$

  Recall that $InProp(W'_{P1})$ belongs both to $InboundSet$ and $OutboundSet$. It leads *to* $Stub(F_{P2})$ and $InProp(W'_{P1})$. $InProp(W'_{P1})$ leads to itself, since inProps and outProps propagate reachability marks in both directions due to the Union Rule. In this case, according to the information stored, it will propagate reachability marks **to** and **from** $OutProp(W_{P3})$. Furthermore, it is not reachable from the GC local-roots of $P1$.

[6]Symbol $\Rightarrow$ means $evaluates\ to$ or $returns$, $\equiv$ relates a field name and its value.

Since inProps and outProps are always implicitly included in their $OutboundFrom$ set, this particular information could have been omitted. Furthermore, the knowledge that inProp and outProp entries always belong to both $InboundSet$ and $OutboundSet$ can also be used to further optimize data representation in snapshots.

For $P2$, we have:

- $Inbound - Set(P2) \Rightarrow \{Scion(F_{P2})_{P2}\}$
- $Outbound - Set(P2) \Rightarrow \{Stub(O_{P3})_{P2}\}$
- $Scion(F_{P2})_{P2} \Rightarrow \{OutboundFrom \equiv \{Stub(O_{P3})_{P2}\}\}$
- $Stub(O_{P3})_{P2} \Rightarrow \{Reach.LOCAL \equiv false\}$

And, for $P3$:

- $Inbound - Set(P3) \Rightarrow \{Scion(O_{P3})_{P3},\ InProp(W_{P3})_{P3}\}$
- $Outbound - Set(P3) \Rightarrow \{Stub(D_{P1})_{P3},\ InProp(W_{P3})_{P3}\}$
- $Scion(O_{P3})_{P3} \Rightarrow \{OutboundFrom \equiv \{Stub(D_{P1})_{P3}\}\}$
- $Stub(D_{P1})_{P3} \Rightarrow \{Reach.LOCAL \equiv false\}$
- $InProp(W_{P3})_{P3} \Rightarrow \{OutboundFrom \equiv \{Stub(D_{P1})_{P3},\ InProp(W_{P3})_{P3}\},$
  $Reach.LOCAL \equiv\ false\}$

The DCD is able to create a DCC-RM containing snapshots form processes $P1..P3$. After performing CMS on the DCC-RM, none of the data structures is marked. This is because none of them is reachable locally to the mutator, and there are no discrepancies among the snapshots, w.r.t. GC and replication data-structures. Such discrepancies, if occurring, would require the adoption of conservative approaches 4..6, when defining the root-set of the CMS, as presented in Section III.2.3.4.3. Thus, the root-set of the CMS, in this case, is an en empty set. Also, it is obvious, from the content of the three snapshots, that there are no reachability marks to actually propagate, since GC local-roots are empty in all processes.

The DCD can thus issue DeleteScion messages regarding $Scion(D_{P1})_{P1}$, $Scion(F_{P2})_{P2}$ and $Scion(O_{P3})_{P3}$ and, issue a DeclareUnreachable message regarding $InProp(W'_{P1})_{P1}$.

In the case of this particular example, the cycle could be broken by sending DeleteScion message to $P2$ regarding $Scion(F_{P2})_{P2}$, and a DeclareUnreachable message to $P1$, regarding $InProp(W'_{P1})_{P1}$

The following sequence of events could take place (others possible):

1. $P2$: $Scion(F_{P2})_{P2}$ is explicitly deleted.
2. $P1$: $InProp(W'_{P1})_{P1}$ is explicitly declared as unreachable.

   From now on, the **DCD is no longer involved** and the cooperation of the acyclic DGC components of each process will reclaim the cycle.
3. $P1$: After the next LGC, an Unreachable message, regarding $InProp(W'_{P1})_{P1}$, is sent to P3. Once acknowledged, the sentUMess bit of the inProp entry is set.
4. $P2$: After the next LGC, $Stub(O_{P3})_{P2}$ disappears, and is not included in the NewSetStubs message to sent to $P3$.
5. $P1$: After the next LGC, $Stub(F_{P2})_{P1}$ is maintained, because it is reachable from the inProp entry and the LGC considers it until it is deleted (i.e., enforces the Union Rule), regardless of the value of sentUMess bit.

   Though maintained, $Stub(F_{P2})_{P1}$ will be ignored if included in a NewSetStubs message received by $P2$. $Scion(F_{P2})_{P2}$ no longer exists, since it has been explicitly deleted, and will not be recreated unless a reference to F_{P2} is exported again to another process. This is clearly impossible with F_{P2} being garbage.
6. $P3$: $Scion(O_{P3})_{P3}$ disappears, after NewSetStubs message is received from $P2$.
7. $P3$: Once the Unreachable message from $P1$ is received (step 3), recUMess bit of $outProp(W_{P3})_{P3}$ is set.
8. $P3$: Later, after the next LGC, object W_{P3} is found to be reachable exclusively from $outProp(W_{P3})_{P3}$, with recUMess bit set. Since the object has not been propagated elsewhere, all the outProps have recUMess bit set. Accordingly, a Reclaim message is sent to $P1$. Once acknowledged, $outProp(W_{P3})_{P3}$ is deleted.
9. $P1$: The Reclaim message is received and $InProp(W'_{P1})_{P1}$ is deleted.
10. $P3$: After the next LGC, $Stub(D_{P1})_{P3}$ disappears, and is not included in the NewSetStubs message to sent to $P1$.
11. $P1$: $Scion(D_{P1})_{P1}$ disappears, after NewSetStubs message is received from $P3$.

With all scions, and inProp/outProp entries removed, the LGC in each process will be able to reclaim all objects and free the memory occupied by them. The distributed cycle comprising replicated objects has thus been successfully detected, broken, and reclaimed.

III.2.3.6 Analysis of Algorithm Properties

In this section, we address the relevant properties of a complete distributed garbage collector discussing them against the algorithm proposed: safety, liveness, completeness, termination, and scalability.

Naturally, since DCC-RM follows an approach analogous to DGC-Consistent Cuts, the arguments presented earlier also apply.

Safety: The acyclic DGC and DCD algorithms are resilient to message loss, delay, re-ordering and replay. Concerning the acyclic DGC, loss or delay of $NewSetStubs$, $Unreachable$ and $Reclaim$ messages does not affect safety because objects deletion is only triggered by reception of these messages. It may, however, delay garbage detection.

Message replay is innocuous since all messages are idempotent. This is trivial for valid $Unreachable$ and $Reclaim$ messages. Invalid messages are discarded because of time-stamp mismatch. $NewSetStubs$ messages always carry information about the most recent scion known when they were first sent. This way, a replayed $NewSetStubs$ message will not prematurely delete scions created more recently.

W.r.t. DCD, when it performs CMS, it is safe when considering scions, inProps and outProps referring to processes that have yet not sent their graph descriptions to the DCD. It conservatively considers them as CMS roots. Furthermore, scions, inProps and outProps, with time-stamps greater than the highest corresponding value known in the process holding the stub, inProp and outProp counterparts, are also conservatively considered as CMS roots.

There are no ordering requirements, and therefore neither competition nor racing conditions, among messages sent to the DCD. Message loss will only delay garbage detection. Replay of older messages may, however, prevent detection of newer cycles. This is solved when an updated graph description is received by the DCD. Additionally, several independent DCDs may execute without error.

Safety in in the presence of concurrency between the cycle detector and the mutator, w.r.t. both object invocation (local), and replication (distributed) is ensured. The arguments for this are analogous to those presented when analyzing the safety of DGC-Consistent Cuts in Section III.2.1.3. Snapshots are assumed to be coherent w.r.t. each process, and the DCD only handles snapshots. Thus there are no race situations between the DCD and ongoing object replications.

Liveness: Algorithm liveness, w.r.t. acyclic distributed garbage, relies on processes sending $NewSetStubs$ messages (containing live stubs) and $Unreachable$ and $Reclaim$ messages regarding unreachable replicas. This is ensured since every process will eventually send these messages after execution of the LGC. The sentUMess bits of inProps are only set when Unreachable messages are acknowledged. The outProps are deleted only when all Reclaim messages are acknowledged. This ensures that the algorithm progresses.

W.r.t. to DCD, the algorithm liveness is obviously dependent on DCD receiving messages, carrying compressed snapshots, by participating processes. Though not required often, processes with mutator activity must eventually update their snapshots. Cycle dismantlement messages must be delivered and acted upon by receiving processes. Indirectly, liveness is also dependent on the acyclic DGC.

Completeness: The algorithm is complete in the sense that any cyclic distributed garbage is eventually encompassed by a DCC-RM, detected and reclaimed. This is achieved creating higher-level DCCs-RM, following the hierarchical approach described (also analyzed in Section III.2.1.3). The replication-aware heuristic adopted in DCC-RM creation further contributes to this.

W.r.t. cycle dismantlement messages, completeness is trivially ensured when all scions and inProps identified as garbage are sent the corresponding DeleteScion and DeclareUnreachable messages. Nonetheless, the DCD is only required to issue messages for a fraction of those (e.g., just one scion), with uniform randomness. In ulterior DCCs-RM, if the cyclic garbage has not yet been broken, different scions or inProps will be targeted, which will hurt the cycle further.

Eventually, all the necessary scions and inProps will receive DeleteScion and Unreachable messages. In worst case scenario, this will amount to send every individual dismantlement message in the first place. After that, the acyclic DGC will delete all its structures and the LGC will reclaim all objects comprised in the cycle.

Floating-garbage consists of just recently created distributed cycles that cannot be detected until snapshots, at processes, correctly reflect it. The algorithm is conservative in these situations. Obviously, this is an inevitable phenomenon to GC in general. However, the relevant issue w.r.t. cycle detection is eventually detecting them, since they are stable (therefore, long-lived), and created at a slow rate.

Termination: Regarding termination, cycle detections are trivially guaranteed to terminate due to the centralized approach used for cycle detection. Once the DCD initiates CMS on a DCC-RM, it will terminate promptly whether cycles are found or not. Propagation of reachability marks during CMS is granted to finish, since it is performed locally and needs to visit each element (belonging to $Inbound - Sets$ and $Outbound - Sets$) only once.

Scalability: The acyclic DGC is scalable because it does not require the communication protocol to support causal delivery, since it is based on DGC-WARM (Sanchez et al. 2001) and reference-listing.

W.r.t. DCD scalability, it stems mainly from the loose synchronization requirements, as cycle detection is performed asynchronously w.r.t. application processes, and snapshot compression. As discussed before, there may be multiple DCDs running, each one creating and analyzing several different DCCs-RM. DCDs can create a DCC-RM hierarchically from existing ones, and cooperate with other DCDs to manage DCC-RM at a higher-level. Furthermore, this hierarchy needs not follow a strict topology.

Summary of Chapter: In this chapter, we presented three novel algorithms for distributed garbage collection, focusing on the detection of distributed cycles of garbage. Two of them, i.e., DGC-Consistent Cuts and Algebra-based Distributed Cycle Detection, are designed for distributed object systems based on remote invocation. Another one, DGC-Consistent Cuts for Replicated Memory (DCC-RM) addresses distributed cycle detection in replicated memory systems, presenting the first viable solution for DGC completeness in such systems.

The three algorithms have some characteristics in common, such as relying on the existence of components for acyclic DGC and LGC in each process, and the goals of safety, completeness, scalability, and asynchrony. They also share a design which minimizes interference with existing support for LGC in virtual machines (i.e., they do not impose a particular approach to LGC), and with applications. Furthermore, they all operate by explicitly deleting DGC structures (e.g., scions) that mislead the acyclic DGC into preserving the distributed garbage cycles. This operation is safe since objects comprised in distributed garbage cycles are already unreachable to applications.

In the first algorithm, DGC-consistent Cuts, distributed cycles of garbage are detected using a centralized approach. A designated server, the distributed cycles detector (DCD), which is asynchronously contacted by application processes, is responsible for constructing DGC-consistent cuts, and detecting garbage cycles enclosed in them. Conceptually, the algorithm constructs a DGC-consistent cut by combining representations of object graphs received asynchronously from application processes. Then, the algorithm performs a conservative mark-and-sweep (CMS) on the DGC-consistent cut.

The second algorithm, proposes a DCD based on a Cycle Detection Algebra (CDA). It employs a de-centralized approach to test whether a cycle candidate indeed belongs to cyclic garbage. Processes forward Cycle Detection Messages (CDM), containing CDA elements. CDMs encode unresolved dependencies (i.e., converging sub-graphs to the cycle being tested), and determine which reference paths should be followed, in order to find out whether they form a distributed cycle of garbage. Cycle detection is initiated by issuing a CDM regarding a scion targeting a suspect-object. If the CDM is forwarded, across a number of processes and in the absence of mutator activity, back to the originating process with all its dependencies resolved, then, a distributed garbage cycle has been found.

The third algorithm, DGC-Consistent Cuts for Replicated Memory (DCC-RM), extends the notion of DGC-Consistent Cuts to distributed systems with data replication, thus serving as a complement to an acyclic DGC for replicated objects, such as DGC-WARM. It operates by creating replication-aware DGC-Consistent Cuts (i.e., DCCs-RM), by combining compressed snapshots received asynchronously from application processes. It detects cycles comprised within them, by performing a conservative mark-and-sweep (CMS), while enforcing the Union Rule during DCC-RM creation, and when performing CMS on them. It does so asynchronously, avoiding any distributed synchronization among the processes comprising the cycles, and without delaying acyclic DGC.

Each algorithm was described in detail in a dedicated section. After a brief intuitive explanation of how each algorithm works, we presented its data structures, messages exchanged, the actual procedure employed for cycle detection, and applicable optimizations (e.g., snapshot compression, hierarchical DCD). Each algorithm was further illustrated with a prototypical example. Finally, we addressed relevant algorithm properties for each algorithm, such as safety, completeness, liveness, termination, scalability.

III3 Implementation

The algorithms presented in the previous chapter were implemented in different environments to demonstrate their feasibility, and independence of a particular run-time. Therefore, the algorithms are applicable to a variety of different scenarios where the notions of reference and reference graph exist. The implementations address the following scenarios:

- .Net Remoting
- OBIWAN
- World Wide Web
- Simulation

The first two scenarios comprise the most important implementation work. They address systems where data is organized in objects graphs (with distribution or replication), and the mutator consists of applications developed with object-oriented programming languages (e.g., Java and C#). The DGC algorithms for distributed object systems were implemented for .Net, and their implementation is described in Section III.3.1. The algorithm for replicated memory systems was implemented on top of OBIWAN (operable in OBIWAN.Net and OBIWAN.Java prototypes), and its implementation is presented in Section III.3.2.

The third scenario addresses the use of distributed garbage collection to ensure referential-integrity, and optimize storage management, in interconnected web sites. It is addressed in Section III.3.3. The fourth scenario, described in Section III.3.4, is used as a test-bed to develop algorithm code free from system-dependent issues, obtain simulation outputs, and perform initial-testing.

III.3.1 Complete Distributed Garbage Collection for .Net

This section describes the implementation of the algorithms presented in Sections III.2.1 and III.2.2 in the context of a distributed object system (.Net Remoting).

The two algorithms assume pre-existence of an acyclic DGC running, e.g., reference-listing. However, .Net does not support distributed garbage collection (DGC). The current approach to DGC in .Net is very simple, featuring only *leases* that are associated with remote objects. When an object X is not invoked remotely for a certain amount of time (bigger than its lease) the reference pointing to X is ignored for reachability considerations; therefore, X may be reclaimed even if there is still a remote reference pointing to it. This means that safety is not ensured. Conversely, if X becomes unreachable before lease expiration, it will not be immediately reclaimed. Thus, leases with long expiration time may create floating-garbage.

III.3.1.1 Acyclic DGC

Therefore, prior to implementing DCD, we require an acyclic DGC in .Net that is both live (as *leases* ensure) and safe (which *leases* are not). This can be achieved using several techniques, with different levels of run-time intrusion. Broadly, these techniques may be characterized as whether they require source-code modifications to, and recompilation of, the run-time.

Since any other distributed cycle detector would require a pre-existing acyclic DGC, we argue that .Net Remoting should have been equipped with one built-in. Such an extension of .Net capabilities is very important for supporting cooperative work among different users, using distributed object-oriented applications. Therefore, we implemented reference-listing, with the extensions described in Section III.2.1.1.1 regarding time-stamps, in the context of Rotor 1.0 (Stutz 2002), officially known as SSCLI,[1] the shared-source version of .Net Common Language Runtime. This work was developed in the context of a project supported by Microsoft (Veiga and Ferreira 2003a).

The implementation of the algorithms can also be accomplished by using other techniques, such as: i) by embedding DGC code in user code (as automatically generated in OBIWAN, and described later), or ii) leveraging extension frameworks provided by the run-time (e.g., sink-chain extension in .Net Remoting using, AOP)[2] as described in (Pereira et al. 2006).

The acyclic DGC (ADGC) algorithm was implemented in Rotor combining modules written in C++ and C#. These languages were chosen to favor integration, simplicity and efficiency. The implementation includes virtual machine modification (for LGC and DGC integration) and Remoting code instrumentation (to detect export and import of references). Virtual machine modifications were implemented in C++. This is the language Rotor core (e.g., LGC) is implemented in. Remoting instrumentation code was developed in C#, since high-level code of the Remoting services is written in this language.

The existing lease mechanism was, for all practical matters, deactivated by modifying file `lifetimeservices.cs`, in order to set lease-lifetime to a minimum value (1 ms), and preventing them from referencing objects.

The data structures representing sets of stubs and scions are implemented in C#, as synchronized hash-tables, for efficiency, and maintained w.r.t. each process running. Code implementing DGC explicit messages is grouped in a specific class `GCManager`. This code runs as a low priority thread in each application process, and is responsible for composing and sending NewSetStubs messages lazily. NewSetStubs messages from other processes are delivered when a well-known remote method made available by class `GCService` is invoked from another process. This method compares stubs received with existing scions, looking for scions whose stub counterpart no longer exists.

[1]**S**hared-**S**ource **C**ommon **L**anguage **I**nfrastructure.

[2]**A**spect-**o**riented **P**rogramming (Kiczales et al. 1997).

III.3.1.1.1 LGC and Acyclic DGC Integration

The ADGC algorithm must cooperate with the LGC, essentially, in two ways:

- the ADGC algorithm must prevent the LGC from reclaiming objects that are no longer reachable locally, but are targeted by incoming inter-process references. This ensures that scions actually prevent objects from being reclaimed.
- the LGC must provide, in some way, the ADGC with information about all remote objects referenced by local objects. This is necessary to ensure that all stubs (representing outgoing inter-process references) are correctly created and preserved.

The first requirement is trivially solved by adding scions to the GC root-set (that already contains GC local-roots). Hash-tables containing scions (and stubs), for each application domain, are maintained as global variables (i.e., referenced by a class static field in `GCManager`). Each scion holds a strong reference to the object targeted by the incoming remote reference.

To fulfill the second requirement, we had to choose between two main options: i) to extend the LGC in Rotor, modifying its implementation, in order to make it able to create and preserve stubs for every outgoing inter-process reference, or at least, provide this information to the ADGC component that would create them, or ii) assuming that stubs are created when outgoing inter-process references are created (e.g., imported), and attempt later to indirectly detect which references have disappeared and eliminate the associated stubs.

The first approach consists in modifying the code of LGC in Rotor so that every field (possibly containing a reference) in every live object is examined. If this reference happens to point to an object that is a transparent proxy, it is an outgoing inter-process reference and the corresponding stub must be created. The second approach consists simply on periodically monitoring existing stubs, verifying that they are still valid, i.e., whether the Rotor transparent proxies associated with them still exist. This is achieved using weak-references, that allow referencing objects without preventing the LGC from reclaiming them.

Both approaches have their advantages and drawbacks. The first option would determine stub deletion more quickly but it could impose larger pause-time on applications, since all stubs would be re-created each time over. Furthermore, this requires implementation using low-level programming (i.e., C/C++ language), in order to minimize this penalty. However, the advantage of determining stub deletion more quickly is mitigated by the fact that ADGC processing is essentially bound to the exchange of messages, and these are sent in a lazy manner, in order not to disrupt running applications.

The second approach was adopted due to the following advantages : i) it does not impose relevant modifications on the Rotor kernel, ii) it can be implemented using a high-level language such as C#, iii) modifications are mainly restricted to the Remoting package, and iv) it does not interfere with the LGC used, and is more general (thus, it can be integrated with any other).

It is implemented via modification of method `CollectGeneration` of class `GCService`, defined in file `comutilnative.cpp`. This is the method, in the Rotor kernel, responsible for performing LGC. The modification consists in inserting, at the end of this method, a call to a static method, `RunDGC`. This method is defined in new class `ClassRunDGC`, placed on file `remotingservices.cs`, under namespace `Remoting`.

The method invocation described is performed from C++ (i.e., native) code, within Rotor core, into C# (i.e., managed) code. This dependency must be registered in file `mscorlib.h` so that, later, the core can obtain a native reference to the C# method. The method invocation must also be properly framed, i.e., a managed code stack must be set-up prior to each invocation. This is achieved using helper macros `HELPER_METHOD_FRAME_BEGIN_0/HELPER_METHOD_FRAME_END`. The invoked method (`RunDGC`) simply delegates execution to an homonymous method defined by the DGC class (`GCManager`). This invocation is not direct because the DGC can only be invoked with the present application domain context already established. This is implicitly achieved by the intermediary C# method that is unique across application domains.

Method `GCManager.runDGC` simply iterates existing stubs, testing if the contained weak-references, targeting transparent proxy objects, are still valid. When they are not, the associated stub is deleted from the hash-table and will not be included in future NewSetStubs messages.

III.3.1.1.2 Remoting Code Instrumentation

Instrumentation of code in Remoting services intercepts messages sent and received by processes when performing remote invocations, so that scions and stubs are created accordingly. In Rotor, messages exchanged by these services are created, intercepted, coded and decoded in several stages, called sinks. A group of different sinks that sequentially process a message constitutes a sink-chain.

Interception only takes place in remote invocations where `MarshallByRef` references are exported or imported. Objects of classes extending `MarshallByRef` are the only ones that can be referenced across application domains (in this case, across different processes). Thus, every time a reference to an object extending MarshallByRef is exported/imported, it must be accounted for DGC purposes.

The reference-listing algorithm demands scions to be time-stamped when they are created and that the same time-stamp is applied to its counterpart stubs. This implies that scion time-stamps must be included in remote invocation messages bearing remote references.

To accomplish this, custom headers were appended to Remoting messages, e.g., *scionIndex*, *machine*, *processId* to uniquely identify DGC structures associated with remote references included in Remoting messages. These values must be propagated throughout the entire sink-chain. Therefore, adaptations were made on base files such as *basetransportheaders.cs, corechannel.cs, message.cs, dispatchchannelsink.cs, binaryformattersink.cs*.

Higher level files such as *remotingservices.cs*, *tcpsocketmanager.cs*, *binaryformatter.cs* and *activator.cs* were also modified primarily to invoke, when remote references are detected, specialized methods included in the previous files.

III.3.1.2 Distributed Cycle Detection

In this section, we describe the relevant implementation details, regarding distributed cycle detection in .Net, of the two algorithms: DGC-Consistent Cuts, and Algebra-based Distributed Cycle Detection. The implementation of the second algorithm reuses code, or otherwise extends it, from the first one, namely w.r.t. snapshot generation and snapshot compression. Therefore, only the novel and different aspects are addressed.

III.3.1.2.1 DGC-Consistent Cuts

Snapshot Generation: Snapshot generation is encoded in C#. In order to be consistent, object graph serialization must be performed while the application code is not running. This is performed using the same technique described earlier. After the LGC, applications are still suspended, since LGC in Rotor is a blocking one. Thus, before allowing the application to proceed, a special method in managed code is invoked (`DGCCuts.GenerateSnapshot`). This method serializes the object graph by serializing a `GCSnapshot` object, that references a series of previously registered objects, in this case, hash-tables containing scions, stubs, and registered application roots (GC local-roots).

Applications need to register their roots so that objects that are reachable from them are also serialized. Otherwise, the snapshot would only include objects reachable from scions. This registration would be unnecessary if thread stacks were considered first class objects in .Net, which are not. This limitation can be obviated by extending the implementation of the Threading package, in order to provide a list of objects referenced from its stack.

Snapshot creation is not performed necessarily after every LGC. It is performed, in this case, just one out of every 100 LGC executions. Object graph serialization is required only for cycle detection. Thus, it only needs to be seldom done. This allows the serialization of the object graph to be done in other more convenient situations, such as when the application stops waiting for input, or is idle. It can be further optimized with operating system support; e.g., using copy-on-write on graph pages. Serialization can then be performed lazily, in the background with minimal delay, extra memory, and processor load.

Snapshot compression: Snapshot compression is coded in C# and has no interaction with the LGC. It is performed, lazily and incrementally, in each process, after a new object graph has been serialized, by a separate thread (which is almost always blocked). Alternatively, compression of snapshots stored persistently in files is performed by an off-line process executing the same function. Snapshot compression is performed by tracing the object graph, using reflection (introspection mechanisms in .Net). This was chosen mainly for simplicity and portability.

A straightforward approach to implement graph summarization is to trace the graph, starting with each scion individually, and determining which stubs are reachable from that scion. An additional trace determines reachability from GC local-roots. This approach may result in each object being traced several times, possibly one for each existing scion.

An alternative approach is to associate a set of flags (implemented as a bit-map) to objects, with length equal to the number of scions plus one to record reachability from GC local-roots. Each bit represents reachability from a specific scion. These bit-maps are updated and propagated while objects are being traced. This trades space (bit-maps) for time (number of object re-tracings).

Initially, all GC local-roots and scions are traced breadth-first, and objects directly referenced from them have the corresponding bit set. All these objects have *depth* one, and are appended to a list of objects to be marked. Each new object found is appended to the end of this list.

Graph summarization progresses, breadth-first, by taking the object at the head of the list and checking, for each object referenced by the former, if its bit-map, as a set, contains all the bits included in the bit-map of the former (possibly being identical). When this is not the case, it means there are new bits that should be propagated (by bit-wise OR) and the latter object is appended to the list for re-tracing.

The previous situation happens when reachability to an object, from a yet unknown scion, has been established. When bit-maps are propagated, the number of object-scanning operations saved, is equal to the number of bits set. Tracing the graph breadth-first maximizes this effect. This way, each object is visited fewer times while propagating more marks (conversely, tracing the graph depth-first would actually nullify savings).

When tracing is finished, the bit-map of each stub is analyzed. The stub is included in the $StubsFrom$ set of the scions, whose corresponding bit is set in the bit-map. $StubsFrom$ sets are also implemented as bit-maps.

Storage occupied by reachability bit-maps can be further optimized in two ways. First, by using an extra indirection that allows reducing storage occupied by multiple, identical bit-maps. This way, objects do not have private bit-maps. Instead, they reference one from a set of active bit-maps. Storage saving is greater in graphs with very high density and confluence of references. Second, in the expectably frequent case of very sparse bit-maps, as the number of scions and stubs grows, bit-maps are replaced by a vector of indexes. We assume an index can fit into 16 bits (i.e., it would allow to uniquely identify 65536 stubs or scions). Therefore, the use of a vector of indexes would provide compression whenever the bit-map has less than 1/16th of its bits set.

Cycle Detector: The DCD is implemented in C# as a separate process. The DCD (server) receives compressed snapshots from application processes (clients) and serializes them to files on disk. The directory of processes and snapshot versions is maintained as a hash-table of lists, whose nodes contain weak-references. When a snapshot is first referenced by a DGC-Consistent Cut, it is loaded from disk. When a snapshot is no longer referenced by any cut, it is reclaimed by the LGC. Its finalizer (i.e., finalization) method determines if the snapshot should be deleted from disk, as newer versions become available. Higher-level snapshots, created hierarchically from a DGC-Consistent Cut, are maintained during a period of time that can be parameterized.

CMS propagates reachability bits from stubs and uses an additional bit to mark visited elements. This ensures termination of the marking process, upon which, garbage scions are identified.

III.3.1.2.2 Algebra-based Distributed Cycle Detection

This algorithms reuses the implementation of DGC-Consistent Cuts, requiring the extensions described in the next paragraphs, and implements the cycle-detection algebra.

In Remoting services, the classes implementing scion and stubs must be extended to hold an additional integer field, the invocation-counter, for DCD-mutator race detection. Invocation-counters must be propagated throughout the entire sink-chain, from scions, to their corresponding stubs. Invocation-counters are included in return messages of remote method invocations. The modifications required are analogous to the ones previously described, in Section III.3.1.1.2.

The code for snapshot compression is reused, handling extended scions and stubs transparently. After the snapshot is compressed, and additional phase is performed to populate $ScionsTo$ sets in stubs. Actually, this information is readily available from the bit-maps propagated to stubs. Therefore, this does not impose any additional time overhead. Stubs require an additional bit-map that may be subject to the optimizations described earlier.

Cycle Detection, i.e., matching of CDA elements, is implemented in C#. Cycle Detection Algebra, carried in each CDM, is serialized using XML. The algebra representation is optimized to minimize redundancy and ease matching. Thus, each scion/stub representation holds two bits, indicating whether they are present in the CDM source and/or target set.

III.3.2 DCC-RM in OBIWAN

The DGC algorithm for replicated memory systems (DCC-RM), described in Section III.2.3, was implemented on top of OBIWAN (operable in OBIWAN.Net and OBIWAN.Java prototypes), within the Memory Management module. OBIWAN.Net can also be run on Rotor, since they are compatible. All the modifications on objects and proxies, required by the DGC, are automatically handled by the OBIWAN compiler, `obicomp` (in its two versions: `obicomp.java` and `obicomp.net`), already presented in Part II.

III.3.2.1 Acyclic DGC

Neither .Net nor Java have a replication-aware ADGC built-in. W.r.t. .Net, it does not include a full ADGC, as already described. Java includes a reference-listing, based on (Birrell et al. 1993a; Birrell et al. 1993b), with *insert* and *remove* messages for reference-list management. It is extended with a lease approach for fault-tolerance (e.g., when a process terminates without sending *remove* messages). Therefore, both acyclic and cyclic DGC components have to be implemented. Nonetheless, data structures and functionality coded for the previous algorithms (Section III.3.1) was reused or extended. This section describes the more relevant differences.

Code implementing acyclic DGC explicit messages is grouped in a specific class *DGCManager* in the Memory Management Module. This code runs as a low priority thread in each application process, and is responsible for composing and sending $NewSetStubs$, $Unreachable$ and $Reclaim$ messages lazily. $NewSetStubs$, $Unreachable$ and $Reclaim$ messages from other processes, are delivered when well-known remote methods made available by DGCManager, are invoked by another process. Messages are sent by performing remote invocations, directly or mediated by the web-bridge (in M-OBIWAN and derived prototypes), of these well-known methods.

Managing Inter-process References: The code for hash-tables containing scions was re-used from Section III.3.1. Each scion maintains a strong reference to the object which is referred by the corresponding incoming inter-process reference. Stubs were extended, and are implemented differently. Their internal weak-reference no longer points to a transparent proxy object, managed by Remoting services. Instead, the weak-reference targets OBIWAN proxy-out objects.

Proxy-out objects encapsulate the means to replicate a specific object from a different process. Therefore they act as inter-process references, in OBIWAN. However, in regular usage, they are used only once, upon which the object is replicated. When proxies-in are created, and proxies-out are de-serialized, i.e., instantiated in a process, they raise events, that are caught by the Memory Management Module, which updates scion and stub information, respectively. This upholds safety rules I and II of the algorithm (*Clean Before Send Propagate*, and *Clean Before Deliver Propagate*, v. Section III.2.3.3.4), by which all references contained in objects being replicated, thus being exported and imported, must be accounted for in DGC structures. When there are multiple proxy-out objects referring to the same object replica in another process (e.g., alternate reference paths), there is only one stub that maintains a list of weak-references.

When a proxy-out disappears, reclaimed by the LGC (because it is no longer referenced, e.g., the object it stood for was already replicated) the weak-reference in the stub becomes invalid. When all proxies-out for the same object have been reclaimed, the stub is eligible for deletion. This code *could* be inserted in the finalizer method of all proxy-out objects. However, deletion of stubs needs only be performed exactly before the next set of stubs is generated, to be sent in a *NewSetStubs* message. Therefore, this operation can be delayed until this moment.

When a new set of stubs is being generated, w.r.t. a process, all weak-references are tested for validity. This technique, using weak-references, is similar to the one described in Section III.3.1.1.1. Processes generate and send NewSetStubs messages after a pre-defined period of time, or number of LGC executions. The occurrence of LGC is detected indirectly, using a *dummy* garbage object. After the LGC, its finalizer method increments a global counter and re-creates the dummy object, in order to detect the next LGC execution.

When a *NewSetStubs* message is received by a process, scions without stub counterpart are deleted, since the corresponding stubs no longer exist (e.g., all proxies-out were resolved). Nonetheless, proxies-in for the objects targeted by the scions are not removed (they may still be used to mediate object update, since they were handed over from the proxy-out to the newly created object replica).

Proxies-in are not needed to prevent replicated objects from premature reclamation, when they are replicated to other processes. The implementation of the Union Rule, described next, ensures this. Moreover, proxies-in are modified to maintain a weak-reference (instead of a normal, strong reference) to the target object. Otherwise, since proxies-in are maintained globally to the process, they would prevent reclamation of all objects, even those that would be otherwise unreachable globally.

III.3.2.1.1 LGC and Acyclic DGC Integration (Union Rule)

The implementation of the Union Rule is greatly simplified if the LGC is able to provide differentiated information regarding object reachability (e.g., from a selected group of objects as InPropLists and OutPropLists). This information allows the DGC to know when it should send *Unreachable* and *Reclaim* messages.

There is no support in existing LGC (both in Java and .Net) for this kind of extended reachability information. On one hand, the ADGC needs to know when the object is no longer reachable to the mutator. On the other, the Union Rule forbids the object replica from being reclaimed. This creates a contradiction to the LGC: *i) detecting un-reachability from local roots results in possible premature reclamation*, while *ii) maintaining additional references to object replicas prevents reclamation, but also precludes the detection mentioned above*. Though it would be possible to modify the LGC (e.g., in Rotor) to provide this information, this would hurt OBIWAN portability, as it would no longer run on a ”standard” virtual machine.

The Union Rule is thus implemented leveraging the limited LGC support available (e.g., weak references, finalizer methods), combined with additions to class extension performed by `obicomp`. Enforcement of Union rule is embedded in object finalizers that call OBIWAN code, as explained now.

InProp and OutProp entries regarding each object, are implemented under its corresponding `OBIRep` structure (as described in Chapter II.3). Conceptually, each InProp and OutProp entry maintains an additional reference to the object replica it refers to; when InProps are created, this reference is a weak-reference. OutProps are created initially containing a strong reference. This addresses issue ii), i.e., that InProp entries should not prevent object local un-reachability from being detected.

We also need to address issue i), otherwise, replicas will be prematurely reclaimed, and the Union Rule violated. This is achieved by the use of finalizer methods. When an object is found unreachable by the LGC, sometime after, the object finalizer is run, if there is one. Thus, when the finalizer is executed, the ADGC knows that the object is not reachable neither from GC local-roots nor from scions (that maintain strong references). However, the object may still be reachable from the InPropLists or OutPropList , and the Union Rule must be upheld. Therefore, the OBIRep is checked for InProp entries. If there are any, the corresponding Unreachable messages are issued, *sentUMess* bits are set, and the weak-reference, contained in the InProp, is promoted to a strong reference, thus *resurrecting* the object replica, in LGC terms. From now on, the object replica, though unreachable to the mutator in the process, will be maintained by the InProp entry, until a Reclaim message is received. After this, there is no additional processing overhead, w.r.t. upholding the Union Rule. When a Reclaim message is received, the corresponding InProp entry is deleted, no longer protecting the object. If there are no additional InProps, the object is reclaimed by the LGC.

Regarding OutProp entries, they initially contain a strong reference to the corresponding object. As a matter of fact, OutProp entries prevent the reclamation of an object until all replicas created from it, are found unreachable in their processes. This is performed by collecting *recUMess* bits. When an Unreachable message is received, the corresponding bit is set. When the *recUMess* bits of all OutProps regarding an object are set, this means that global reachability of all them is dependent exclusively on this replica: if it is reachable, all other should be preserved; otherwise, all replicas, including this one should be deleted. This check is performed by *demoting* the strong references in such OutProps, to weak-references. If the object replica is reachable from GC local-roots or from scions nothing happens. Otherwise, its finalizer will eventually be executed. The execution of the finalizer method will verify that there are OutProps, and for each one, it will issue a Reclaim message, and delete the OutProp. After this, the object is at the mercy of the LGC, as all other replicas will be in their processes, when Reclaim messages are acted upon.

Recall that the combined code, checking InPropLists and OutPropLists is actually encapsulated in a special method invoked by object finalizers. This invocation is automatically inserted by `obicomp`, thus minimizing intrusion with application code.

Handling Re-propagation of an Object Replica: One issue separates OBIWAN.Net and OBIWAN.Java prototypes: when object replicas previously deemed as unreachable locally, become reachable to the mutator again. This can happen if the process holding a replica that is unreachable locally, has references (direct or indirectly) to another replica of the same object in other process (possibly the same where the object was initially replicated from). In such a situation, a process may hold a InProp and a stub regarding the same object. One regards object replication, while the other indicates there are other inter-process reference(s) targeting the object. When these references are followed, i.e., proxy-out objects are invoked, the proxy-out is resolved and, according to the replication mode, either the local object replica is promptly used, or it is refreshed with the contents of the object replica of the other process. Either way, the object replica has become reachable to the local mutator again. This is signaled by resetting sentUMess/recUMess bits and updating time-stamps, as described in Section III.2.3.3.3.1, and resetting weak/strong references accordingly.

This poses a problem: how will local un-reachability of this object replica will be detected in the future, to uphold the Union Rule. This depends on the possibility of executing object finalizers more than once. In .Net, this is possible. Therefore, method `GC.ReRegisterForFinalize` is invoked when re-propagation occurs. This way, it is possible to simply demote the strong reference contained in the InProp (protecting the object replica until reception of Reclaim message) to a weak-reference (re-installing the mechanism to detect local un-reachability in the future).

In Java, object finalizers may be run only once. Therefore, the same solution cannot be applied, and an alternative approach is followed. When a replica, previously found as unreachable locally, is re-propagated, there is no object that is referencing it; otherwise, it would not be unreachable locally. This way, it is safe to reconstruct the object, that is, to instantiate a new object, install it in the same OBIRep entry, and update it with the content of the existing replica (with its references to object replicas, replaced with references to proxy-out objects)[3] or another one

[3]This prevents the replica being reconstructed from possibly referencing an object whose finalizer method has

(if the local one is being refreshed in the process). When proxies-out, regarding this replica, are resolved they will invoke and return references to the new replica. Later on, when the replica is found to be unreachable locally, its finalizer will be executed as expected, upholding the Union Rule. The cost of this object reconstruction is only payed when a replica is re-propagated. Thus, while not null, it can be amortized by communication costs, that are orders of magnitude greater.

Since in both cases, finalizers regarding the same object replica (in logical terms) may be run several times, pre-existing finalizer code defined by the application, is still executed only once by testing a flag that is kept in the OBIRep.

III.3.2.2 Cycle Detection

DCC-RM reuses C# code for generating and compressing snapshots. Snapshots are generated by serialization (reusing method `DGCCuts.GenerateSnapshot`) while providing an extended set of roots, comprised of scions, stubs, InProps, OutProps, and registered application-roots (GC local-roots). In the current implementation, snapshots are generated while the mutator is otherwise stopped, i.e., not manipulating replicated objects. This is achieved by direct application command (via an utility method, `SnapshotCheck` provided by OBIWAN). This method simply checks a flag and generates the snapshot if it is set, resetting it after. The flag is set by OBIWAN code, in the current implementation, after a parameterizable period of time expires, and only if there have been changes to replication or DGC structures.

Snapshot compression code is reused by providing the elements of the $InboudSet$ in the place of scions, and the elements of the $OutboundSet$ in the place of stubs. Because the sets share elements (inProps and OutProps), the resulting compressed snapshot is compliant with the Union Rule, and DCC-RM specification.

Tracing is performed as described in Section III.3.1.2.2, and the same optimizations apply. It starts from GC local-roots, and elements in the $InboudSet$, regardless if they are scions, InProps, or OutProps. All objects conceptually have an associated bit-map (wide enough to hold one bit per each existing scion, InProp, OutProp, plus one bit to account for reachability from GC local-roots). Bit-maps are propagated towards stubs, InProps, and OutProps. When one of them is reached, tracing along that path stops.

The DCC-RM cycle detector is extended to adhere to the special heuristic described in Section III.2.3.4. CMS code is reused, upon which garbage scions and InProps are identified.

III.3.3 Referential Integrity and DGC in Web systems

This section presents the main aspects of the application of DGC algorithms to web systems, partially described in (Veiga and Ferreira 2003c; Veiga and Ferreira 2004b). It addresses the use of DGC algorithms to enforce referential integrity and perform storage management in: i) web

already been run (an object which already triggered an Unreachable message). When the proxy-out is transversed, if the referenced object has already been finalized (because an Unreachable message has been sent w.r.t. it), it will be reconstructed, as described, and the proxy-out will return a reference to the new object.

systems, with replication of static content (Veiga and Ferreira 2003c), and ii) with dynamic content generation, without replication (Veiga and Ferreira 2004b). These works, combined with DCC-RM, can achieve completeness w.r.t. memory management, in the presence of both replication and dynamically generated content. All content replicated and/or dynamically generated, reachable from a definable root-set (e.g., *bookmarks*), is preserved. Unreachable content is reclaimed.

III.3.3.1 Background

There are no large-scale mechanisms to enforce referential integrity[4] in the WWW; broken links prove this. For some years now, this has been considered a serious problem of the web (Lawrence et al. 2001). This applies to several types and subjects of content, e.g., i) if a user pays for or subscribes some service in the form of web pages, he expects such pages to be reachable all the time, ii) archived web resources, either scientific, legal or historic, that are still referenced, need to be preserved and remain available, and iii) dynamically generated content should also be accounted and it should be possible to preserve different execution results with timing information.

There are several techniques useful in finding web pages with similar content (Lawrence et al. 2001). Nevertheless, if the user has payed for, or subscribed to this service, he will consider this broken-link failure as a breach of the service, contracted or agreed upon. So, this situation, apparently and regularly innocuous in more un-formal domains, can undermine content providers reputation and imply severe losses and costs (Ingham et al. 1996).

As serious as this last problem, there is another one related to the effective loss of knowledge. As mentioned in earlier works, broken links on the web can lead to the loss of scientific knowledge, further aggravated as web contents get older (Lawrence et al. 2001).

Nevertheless, solutions that try to preserve every and anything can lead to massive storage waste. Therefore unreachable web content, i.e. garbage, should be reclaimed and its storage space re-used.

III.3.3.1.1 Dynamic Content

The weight of dynamically generated content versus static content has progressed enormously. From a few statically disposed web pages, the WWW now encompasses millions of dynamically generated pages, resorting to user context, customization, user class differentiation. Today, the vast majority of web content is dynamically generated, shaping the so-called deep-web, and this has been increasing for quite some time now (O'Neill et al. 2003; Bergman 2001). This content is frequently perceived by users as more up to date and accurate, therefore having more quality. Since this content is generated on-the-fly, it is potentially different every time the page is accessed. This may be due to different query parameters, different server and session state, or simply because it changes with time.

[4]Preservation of every resource, still targeted by references, contained in other resources accessible from some defined root.

It is clear that this type of content cannot be preserved by simply preserving the script files that generate it. This is specially relevant with content changing over time. It is produced by script pages that, although invoked with the same parameters (identical URL), produce different output, at every invocation, or periodically. Examples of these include stock tickers, citation rankings, ratings of every kind, stocks inventories, so called last-minute news, etc. So, changes in produced output, or in the underlying database(s), should not prevent users from preserving content of interest to them, and keep easy access to it.

Although data is only lost when actual data sources (e.g., database records) are deleted, it could become otherwise unavailable or inaccessible causing effective loss of information. Thus, dynamic content, in itself, must also be preserved while it is still referenced and not just the script pages that generate it. Furthermore, other pages pointed by URLs included in every reply of these dynamic pages must be preserved, i.e., content dynamically referenced must be also preserved while it is reachable.

III.3.3.1.2 Current Approaches

The task of finding broken links can be an automated using several applications (HostPulse 2002; LinkAlarm 1998; XenuLink 1997). However, these applications do not enforce referential integrity because, while useful in detecting local and remote broken links, they cannot neither prevent them from occurring, nor reclaim wasted storage.

Current solutions to the problem of referential integrity in the web are unable to cumulatively address the following issues:

- **replication**: they are unable to handle data replication correctly (Creech 1996; Ingham et al. 1995; Ingham et al. 1996; Moreau and Gray 1998), or impose strict replica consistency (Andrews et al. 1995; Kappe 1995).
- **dynamically-generated content**: they neither preserve different versions of content generated from a dynamic web page, nor other documents referenced from it (Rosenthal and Reich 2000; Reich and Rosenthal 2001; Creech 1996; Andrews et al. 1995; Kappe 1995; Ingham et al. 1995; Ingham et al. 1996; Moreau and Gray 1998).
- **storage reclamation**: they either: i) preserve every content produced, regardless of being reachable from a specifiable root-set (Rosenthal and Reich 2000; Reich and Rosenthal 2001; Creech 1996; Andrews et al. 1995; Kappe 1995), or ii) when they include DGC, they are not complete, i.e., they are unable to detect and reclaim distributed cycles of garbage comprising web content (Ingham et al. 1995; Ingham et al. 1996).
- **transparency w.r.t. existing web infra-structure**: they impose specific authoring, visualization and administration schemes to the web (Creech 1996; Andrews et al. 1995; Kappe 1995; Ingham et al. 1995; Ingham et al. 1996).

In summary, some systems currently supporting replication, do not address referential integrity, and consider broken-links as system or application failures that should be exposed to users. Some solutions addressing the problem of referential integrity either do not take replication, or dynamically generated content, or storage management, into account.

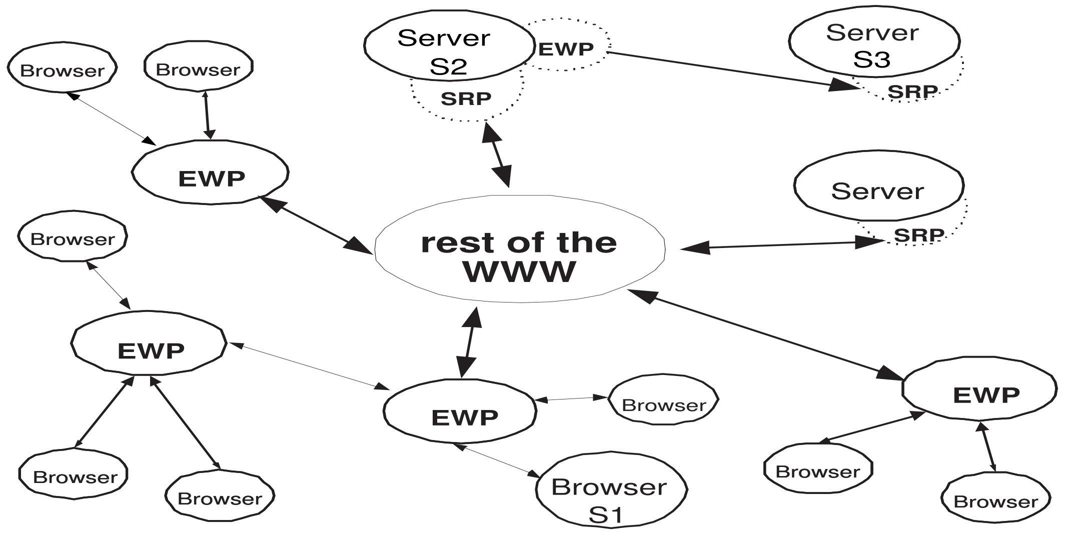


Figure III.3.1: *General architecture of system deployment.*

III.3.3.2 Web Architecture and DGC Integration

In order not to impose the use of a new, specific, hyper-media system, the architecture proposed is based on regular components used in the WWW or widely accepted extensions to them. The system is designed following OBIWAN architecture (presented in Chapter I.2) mapped to the web scenario, exemplified in Figure III.3.1. It comprises the following entities:

- web servers.
- clients - web browsing applications.
- extended web-proxies (EWPs) - these manage clients requests and mediate access to other proxies.
- server reverse-proxies (SRPs) - intercept server generated content and manage files.

The adaptations of the OBIWAN network architecture (portrayed in Figure I.2.2) to the web scenario (depicted in Figure III.3.1) are the following. Server reverse-proxies (SRPs) and extended web-proxies (EWPs) act as middleware layers, for replication and memory management purposes. Browsers perform the role of client applications, and servers perform the role of object repositories in OBIWAN. The main differences stem from the fact that the execution environment of web-proxies and web-servers is considered opaque, since modifying them would hinder the portability of the system. Therefore, all middleware data structures and functionality is maintained out of them, in separate processes (i.e., SRPs and EWPs). In OBIWAN prototypes for Java and .Net, middleware structures (e.g., regarding replication) are maintained within running application processes.

The entities manipulated by the system are web resources in general. These come in two flavors: i) HTML/XML documents (or pages) that can hold text and references to other web resources, and ii) all other content types (images, sound, video, etc.). Resources of both types can be accessed, and replicated. The system ensures they are preserved while they are still reachable. HTML/XML documents can be either static or dynamically generated/updated. Other web resources, though possibly dynamic as well, are not considered w.r.t. references to other resources and are viewed, by the system, as leaf-nodes in a web resources graph. Thus, memory is organized a replicated memory system (described in Section III.2.3.2) comprised of web resources connected by references (in the case of the web, these are URL links).

Static HTML/XML and other media (e.g., binary files) content may be freely replicated and made to diverge by any authoring tool (e.g., versions of same document in different languages). The Union Rule preserves all replicas while at least one of them is reachable. Propagation paths allow navigation through replicas of the same document. Different versions of dynamically generated content, produced by the same dynamic page, are also preserved. When dynamically generated pages are replicated, the content previously generated is copied into the new replica.

We consider, mainly, as cases of web usage:

- i) web browsing without content preservation , i.e., standard web usage.
- ii) web browsing, possibly with replication, with book-marking desired explicitly by the user, either in a page-per-page basis or transitively. This enforces referential integrity, preserving all content reachable from the bookmark-set (i.e., the root-set).

A typical user in (e.g., at $S1$) browses the web, accesses and bookmarks some of the pages from, for example, web-server (e.g., at site $S2$) (see Figure III.3.1). Nothing prevents running the extended web-proxy (EWP) in the same machine as the web browser, though it would be obviously more efficient to install a proxy hierarchy. From the user point of view, the client side of the system is a normal web browser with an extra toolbar. This toolbar allows book-marking the current web page (or a URL included in it) as a member of its root-set, and inform the proxy of such. It also allows the user to replicate web resources (e.g., a HTML file) into his computer, i.e., to create a stable local replica of the resource he is looking at. Replication is nearly inexpensive w.r.t. time, taking advantage of the fact that file being browsed may be, implicitly, already available in the local system cache.

Once book-marked, these documents may hold references to other (not book-marked) web resources in site $S2$. Thus, it is desirable that such resources in site $S2$ remain available as long as there are references pointing to them. Web resources in other servers (e.g., site S3), targeted by URLs found in content from site $S2$ are also preserved, while they are still referenced. The system ensures that web resources in sites remain accessible, as long as they are pointed from some HTML/XML document. In addition, web resources, which are no longer referenced from any other document, are automatically deleted by the garbage collector. This means that neither broken links nor memory leaks (storage waste) can occur.

Figure III.3.2 presents an example web graph, with dynamically generated web content (the two dynamic URLs on the left-hand side of the figure) preserved several times, represented like pages over pages. These preserved dynamic pages hold references to different HTML/XML documents, depending on the time (and session information) when they were book-marked. Preserved dynamic content is always stored at the SRP to maintain transparency w.r.t. the server.

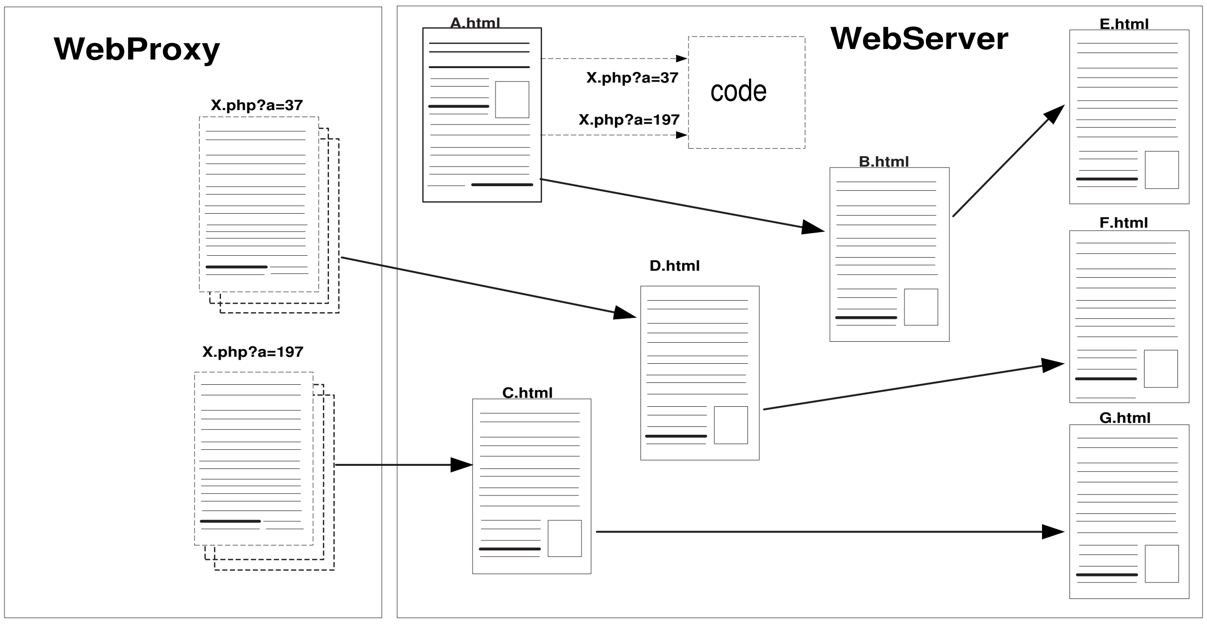


Figure III.3.2: Example web graph with several versions of previously dynamically generated content.

III.3.3.3 Deployment

The prototype implementation was developed in Java. It deploys a stand-in proxy that interprets HTTP-like custom requests to perform DGC operations and relies on a ”real”, off-the-shelf web-proxy, running on the same machine, to perform everything else.

Preserving dynamically generated content raises a semantic issue about browser, proxy and server behavior. When a dynamic URL, previously preserved, is accessed, two situations can occur, depending on session information shared with the proxy: i) the content is retrieved as a fresh execution , or ii) the user is allowed to decide, from previously accessed and preserved content, which one he wants to browse. W.r.t. replication, a user may also want to navigate to other replicas of the same document, following InProps/OutProps as implicit references.

The selection of these alternatives in navigation is implemented by the middleware code in EWPs and SRPs. It allows two configurable default behaviors:

- when a URL regarding a dynamic or replicated web page is requested, the browser receives an automatically generated HTML reply, with a list of previously preserved content, including other replicas it knows about, provided with date and time information.
- the very HTML code, implementing the link to the mentioned URL, is replaced with code that implements a selection box, offering the same alternatives as the first option. The first behavior is less computationally demanding on the proxies but the second one is more versatile, in terms of user experience.

W.r.t. dynamically generated content, SRPs perform URL translations to access corresponding files that hold the actual preserved content. As portrayed on the left-hand side of Figure III.3.2, multiple such files may exist, preserving different contents generated by the same dynamic page.

Local GC: The local garbage collector (LGC) is responsible for eventually deleting or archiving unreachable web content. It must be able to crawl the server contents. The root-set of this crawling process is defined at each site; it must include scion, InProp, and OutProp information provided by the DGC.

Crawling is performed only within the site and lazily, in order to minimize disruption to the web server. The crawler maintains a list of documents to visit. They are parsed and references, found within them, to documents in the same server, are added to this list. When dynamic content is found, the results of previous executions are all traced. References found are saved in auxiliary files. These can be re-used later by the crawler, in another crawl, if the document was not modified.

When it is necessary to update web page content (a modification to static or programmatic page), it will be locked and signaled as the crawler must, for safety reasons, re-analyze it. This is performed following the links included in both versions of such web-page (the previous and the new one). Then, after the whole local graph has been analyzed, the new version of the DGC structures replaces (flip) the previous one. Unreachable web documents and other resources can then be archived or deleted.

Web resources can be created with any authoring tool. Once created in some site, web resources must become reachable in order to be accessible for browsing. This can be done in two ways: i) add a reference to the new resource in the local root-set, or ii) add a reference to the new resource in some existing and reachable document. New documents and resources always survive at least one GC execution before they can be reclaimed.

Acyclic DGC: The distributed garbage collector is responsible for managing inter-site references and replicated web resources. Each EWP and SRP enforce the safety rules of the algorithm, parsing the content transferred to create the appropriate GC data structures.

The sending of DGC messages is implemented by invocation of Java *servlets*, at the destination server. Client-only sites receive DGC messages piggy-backed in replies from servers. The DGC component in each site acts upon received messages, possibly deleting scions, InProps, and OutProps.

Cycle Detection: Code for snapshot creation reuses the web crawler responsible for LGC. It incrementally copies all site content to a different location (e.g., directory tree, disk partition). During snapshot creation, all page modifications must be recorded and modified pages must be also copied and re-traced. Unmodified pages need not be re-scanned.

The location holding the snapshot contains a conservative view of the site. It may have some files that are no longer reachable (e.g., they were referenced from files that were since modified). This improves liveness w.r.t. snapshot creation. It does not hinder completeness, since the DCD is only used for detecting cyclic garbage, not acyclic or local. Code for snapshot compression only handles snapshots already completed, and produces compressed snapshots in the format used by DCC-RM in OBIWAN.Java.

III.3.4 DGC Simulator

The three algorithms were initially implemented in the context of a more general DGC simulator, developed in Lisp, used in GC-Portal (Ferreira and Veiga 2005), that allows easy prototyping of DGC algorithms, without the constrains of a particular environment, communication libraries, or programming languages. It is defined in a main module, `gc-lib.lsp`, that may be used or extended by other modules (e.g., `cda-dcd.lsp`, for Algebra-based Cycle Detection).

III.3.4.1 Module `gc-lib.lsp`

The `gc-lib.lsp` module provides a base representation of distributed object graphs. Objects are abstracted as an opaque payload and a list of references to other objects, possibly in other processes. This representation of distributed objects graphs includes processes, objects, LGC and DGC structures. It provides an implementation of simple DGC algorithms: reference-listing (with time-stamps), and global and sequential tracing. LGC is performed atomically using tracing, since the main goal of the simulator is to study the distributed aspects of GC, namely distributed cycle detection. Reference-listing can be reused when implementing specialized cycle detectors that operate on top of it (e.g., DCD based on backtracking, migration, cycle-detection algebra).

W.r.t. simulating communication among processes (e.g., remote invocations and DGC messages), all processes have two message queues, one for inbound and the other for outbound messages. All interaction among processes is performed exclusively by posting and retrieving messages in queues. In each iteration, the simulator randomly selects a queue of a process and either transfers (for outbound queues), or delivers (for inbound) the first message in it. This ensures non-determinism, and allows interleaving with distributed invocations (explained next) to examine algorithm safety and reproduce race conditions.

Messages follow a life-cycle of three phases: i) send (placed on outbound queue), ii) transfer (moved to the inbound queue of the receiving process), and iii) deliver (fetched from the inbound queue and acted upon in the receiving process). Messages are implemented as lists whose first element is a function to be invoked in the context of the receiving process (i.e, whose data structures are passed as parameters to the intended function), when the message is delivered.

The mutator is simulated by resorting to messages. These messages contain additional data: i) a target process, ii) a target-object, and iii) a path-list (which is a list of numbers that represent positions of reference-fields to transverse in objects). When a process receives a mutator-message, it analyzes the enclosed target-object and path-list. Each element in the path-list specifies a reference, by position, to be transversed from the object being ”invoked”. When a reference corresponding to a field position is transversed, it is removed from the path-list (its head), and the remaining of the list (its tail) is applied to the referenced object. This continues until a `nil` reference is selected, or the path-list becomes empty. Optionally, The last object referenced may be returned as a result, have one reference appended, or modified.

Each time the simulated mutator needs to cross a process boundary, it stops transversing objects, and queues a message containing the target process and object, and the remaining of the path-list that was not used. Upon message delivery at the destination process, that may happen some time after, the distributed invocation continues. The same technique is used to propagate return values of distributed invocations. The simulated mutator is rather general, since it ensures that messages related to distributed invocations are completely asynchronous w.r.t. messages exchanged by the DGC algorithm.

III.3.4.2 Module `cda-dcd.lsp`

Algebra-based Cycle Detection is implemented as message forwarding and CDA matching, defined in module `cda-dcd.lsp`, on top of the simulated runtime provided by `gc-lib.lsp`. Snapshot compression converts the object graph in each process into a list of stubs and scions. Each scion and stub is extended with StubsFrom and ScionsTo lists, without any optimization.

CDMs are exchanged as queued messages. Upon delivery, the CDA matching function is invoked against the snapshot of the receiving process. If there are any CDMs to be forwarded, they are appended to the outbound queue of the process.

III.3.4.3 Module `problem.lsp`

Tests are encapsulated as *problems* that comprise a distributed graph and a DGC specification that includes a DGC algorithm and, optionally, a cycle candidate. This module also defines functions to create and compose test graphs, based on common topologies, such as linked-lists, doubly-linked lists, trees, etc.

All steps performed by the simulator are sequenced for demonstration and debug. The main sequencer randomly selects messages queues, as already described, for message transfer and delivery. When there are no more queued messages, or a cycle has been detected, the sequencer terminates.

III.3.4.4 Visualization Tool

The simulator is able to export snapshots (plain and compressed) both in XML and using Lisp serialization (*parenthesis-serialization*). For parsing, only Lisp serialization is used. XML serialization is used to interface with a visualization tool, developed in C# that allows the design, update, and modification of distributed object graphs. When the visualization tool is used, the GC simulator is executed, performs only one step (LGC, message deliver, etc.), and exports the distributed graph again for visual update.

The C# application is able to parse XML representations of distributed graphs, to update its internal state. It exports distributed graphs, using XML for storage, and using Lisp serialization, implemented in C# (method `toLisp()` analogous to `ToString()`), to feed the simulator.

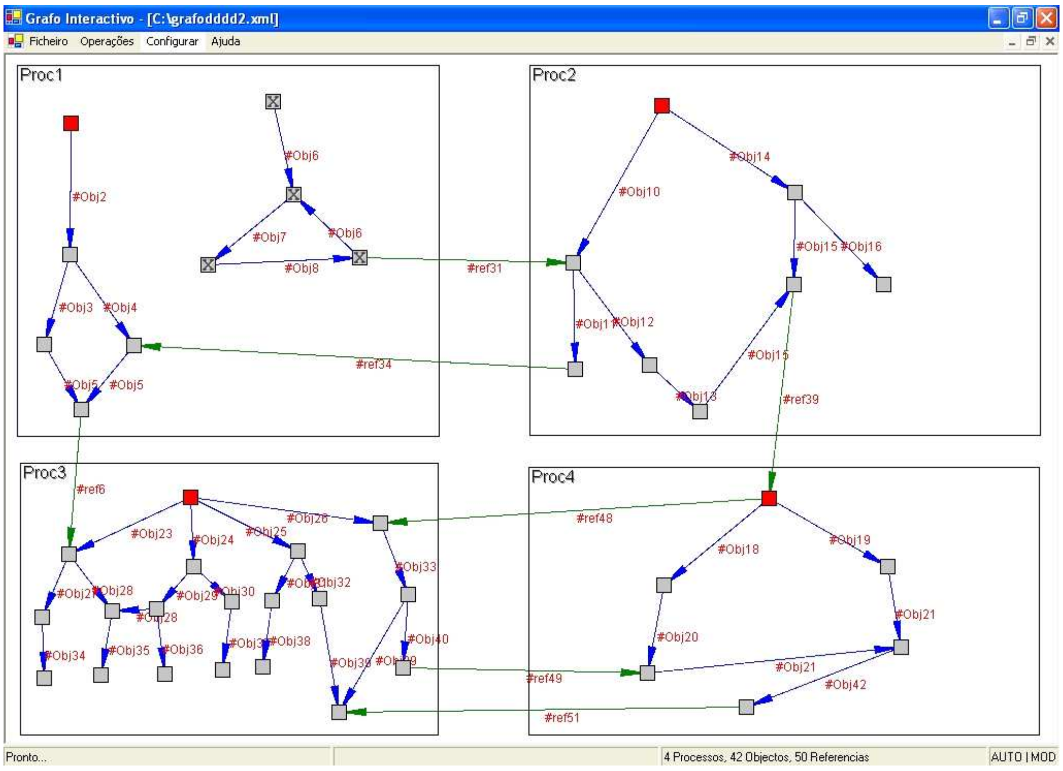
Figure III.3.3 shows a typical interaction with the visualization tool. The top screen-shot displays a distributed graph, spanning four processes, where some objects have been identified as garbage, in the top-left process. The bottom snapshot shows the same distributed graph, after interaction with the simulator, where those same objects have already been reclaimed.

Summary of Chapter: In this chapter, we presented the most important aspects of the implementation of the three algorithms proposed in the previous chapter, in a number of scenarios where the notion of referential integrity is applicable.

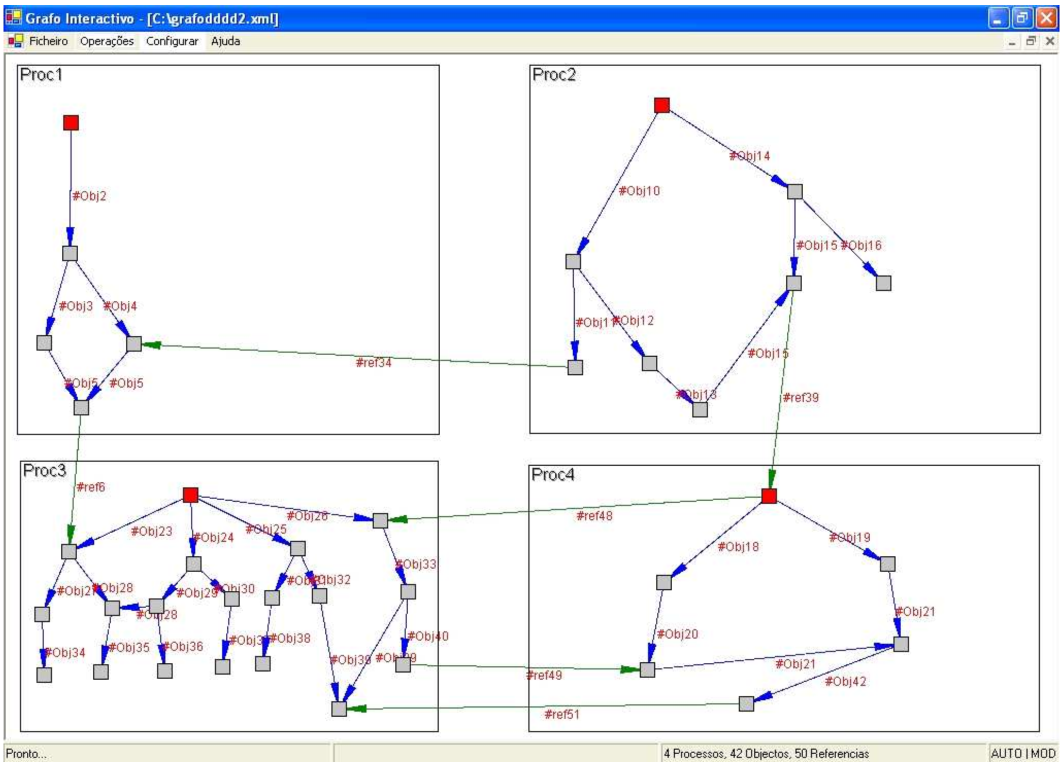
First, we described the implementation of complete DGC for .Net Remoting, in the context of the Rotor virtual machine (a shared-source version of .Net). Prior to the specifics of each algorithm for distributed cycle detection, we presented the relevant details concerning the implementation of a full acyclic DGC in Rotor (since the built-in DGC approach is not safe), and the integration with its LGC. The necessary modifications to the Rotor VM, which are mostly restricted to the Remoting package, were also described. They deal essentially with monitoring inter-process references being exported and imported during invocation of remote methods. We then described the common aspects of implementing cycle detection (i.e., snapshot creation and snapshot compression), as well as some details on higher-level code, responsible for the creation of DGC-consistent cuts, and the processing of the Cycle Detection Algebra.

Regarding complete DGC in replicated systems, we presented the implementation of DCC-RM in the context of OBIWAN. We described how acyclic DGC rules are implemented, resorting to code in proxy objects and finalization methods. We addressed the re-propagation of object replicas in detail and presented solutions for the limitations, w.r.t. this purpose, found in existing LGC for the Java and .Net virtual machines. The techniques employed were object reconstruction and re-register finalization. The whole implementation requires no changes to existing Java and .Net VM, resorting exclusively to middleware code. It serves (together with AOP-DGC) also as a demonstration of the portability of DGC-Consistent Cuts and Algebra-based DCD, although having been implemented via extension of the VM.

We also described the main aspects concerning the implementation of DGC algorithms for web systems. In this work, the emphasis was on handling dynamically generated content and integration with existing web infrastructure (e.g., browser, servers, caching), without imposing modifications to it. Finally, we presented the relevant aspects of a Lisp-based DGC simulator and associated visualization tool, used during algorithm development and testing.



Pronto... 4 Processos, 42 Objectos, 50 Referencias AUTO | MOD



Pronto... 4 Processos, 42 Objectos, 50 Referencias AUTO | MOD

Figure III.3.3: Simulator appearance: Before and After LGC.

III.4 Evaluation

This chapter presents the evaluation of the DGC algorithms presented in Chapter III.2. This evaluation focus quantitative as well as qualitative aspects. W.r.t. quantitative ones, we present the performance results concerning the most relevant issues of the described implementations. Regarding qualitative aspects, we present a comparison between the proposed algorithms and existing solutions found in the literature. This stems from the discussion offered in Sections III.2.1.3, III.2.2.3, and III.2.3.6. We conclude with an evaluation of proposed and existing solutions w.r.t. portability, i.e., the level of run-time intrusion and coupling imposed by each GC solution.

The rest of this chapter is organized as follows. The next section describes the most relevant performance results regarding the implementation of the three DGC algorithms, presented in Section III.2. The results are focused on the following aspects: i) overhead of acyclic DGC rules, ii) snapshot creation, iii) snapshot compression, and iv) enforcement of Union Rule in DCC-RM. In Section III.4.2, we present a brief evaluation of DGC in web-systems, both in terms of performance impact, as well as usability and integration. Section III.4.3 offers a comparative evaluation of the three algorithms proposed in Section III.2, w.r.t. relevant related work in their respective scenarios (distributed object systems in the case of DGC-Consistent Cuts and Algebra-based DCD, and replicated memory systems w.r.t. DCC-RM). Finally, in Section III.4.4, we offer an analysis, regarding the portability of GC solutions to existing virtual machines, encompassing both relevant related work and our own.

III.4.1 Performance Results in .Net and OBIWAN

The most relevant performance results of the DGC implementation are those related to phases critical to applications performance: i) enforcement of acyclic DGC safety rules, and ii) snapshot serialization. These phases are those that could delay and potentially disrupt the mutator, therefore applications.

DGC safety rules impose the creation of stub/scion pairs when inter-process references are exported and imported. They are common to any acyclic DGC. This occurs when references are passed as parameters or results of remote method invocations, in distributed object systems such as .Net Remoting. Similarly, in replicated memory systems, such as OBIWAN, references are also imported/exported, when contained in objects that are being replicated. In OBIWAN, stub and scion creation is uphold by Safety Rules I and II.

We also present a performance study regarding snapshot compression, which can be performed off-line. Therefore, we focus on the potential savings, w.r.t. memory and network bandwidth, provided by this technique.

W.r.t. the implementation of DCC-RM in OBIWAN, we analyzed the potential overhead resulting specifically from the enforcement of the Union Rule of DCC-RM. Since it may also cause delay to the mutator, though only in specific situations, it is thoroughly measured and addressed, both in Java and .Net implementations.

III.4.1.1 Overhead of acyclic DGC rules

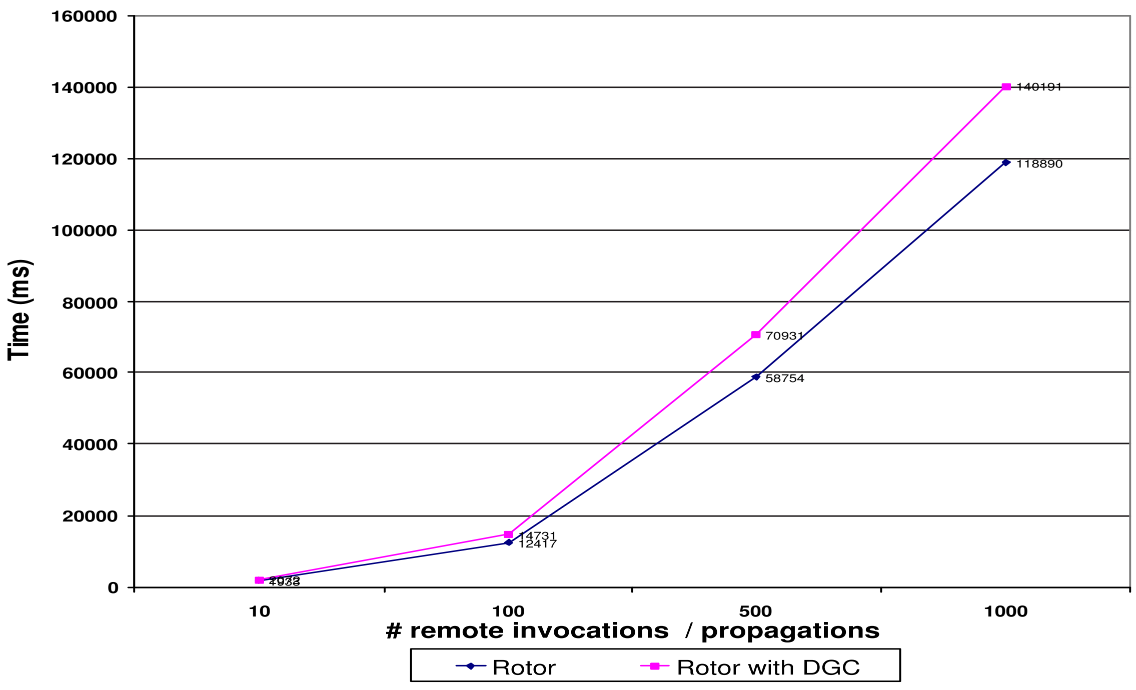
We measured the creation of stubs and scions when remote references are exported/imported during remote invocations, in Rotor. These operations are always performed and cannot be fulfilled lazily. We tested worst case scenarios, discarding potentially long network communication times, that could mask stub/scion creation overhead. The results were obtained using a Pentium 4 Mobile 1600Mhz with 512 Mb RAM.

Figure III.4.1 shows results for increasing series of remote invocations of a remote method, with 10 arguments (10 different references being exported/imported), where client and server processes execute in the same machine. This forces the DGC to create 10 scions and stubs each time the remote method is invoked. The overhead associated with the creation of stubs and scions is evaluated in a worst-case scenario since: i) processes are running on the same machine therefore without network communication delay, ii) there is no additional payload (e.g., serialized objects) in messages besides inter-process references, and iii) the remote method being executed is *empty*, without any actual processing. Performance penalty is within 7%-21% which is acceptable for the functionality provided, i.e., a safe DGC (not a lease-based one) running on Rotor.

The overhead is due to the creation of so may inter-process references, in the last-case, 10000 references. Nonetheless, the overhead stays consistently below 2.5 ms per reference being imported/exported during remote invocation. This is confirmed by the maximum overhead observed (1.218 ms) w.r.t. each reference interception (on the server-side to create scions, and on the client-side to create corresponding stubs). In the presence of network communication, this overhead would be mostly masked by network latency.

AOP-based DGC: The interception of inter-process references being exported/imported during remote method invocations may be performed, in .Net, without modifying the virtual machine. The work in (Pereira et al. 2006) describes the implementation of DGC-Consistent Cuts, using the aspect-oriented features provided by .Net (class `ContextBoundObject`), to extend .Net Remoting. Objects from classes inheriting from `ContextBoundObject` are decorated with attributes that trigger DGC code whenever they are invoked, or references to them are transferred between processes.

The increased portability comes with a cost, nonetheless. In the same worst-case scenario described before, the performance penalty on object invocation has an average of 19.25% (which is nil in Rotor), while reference import/export has an average penalty of 40% (compared to at most 21% in Rotor). Again, these are results for the worst-case scenario described.



# of invocations	Rotor	cost per operation	Rotor w/ DGC	overhead per operation	Variation
10	1933	9.665	2072	0.695	7.19%
100	12417	6.209	14731	1.157	18.64%
500	58754	5.875	70931	1.218	20.73%
1000	118890	5.945	140191	1.065	17.92%

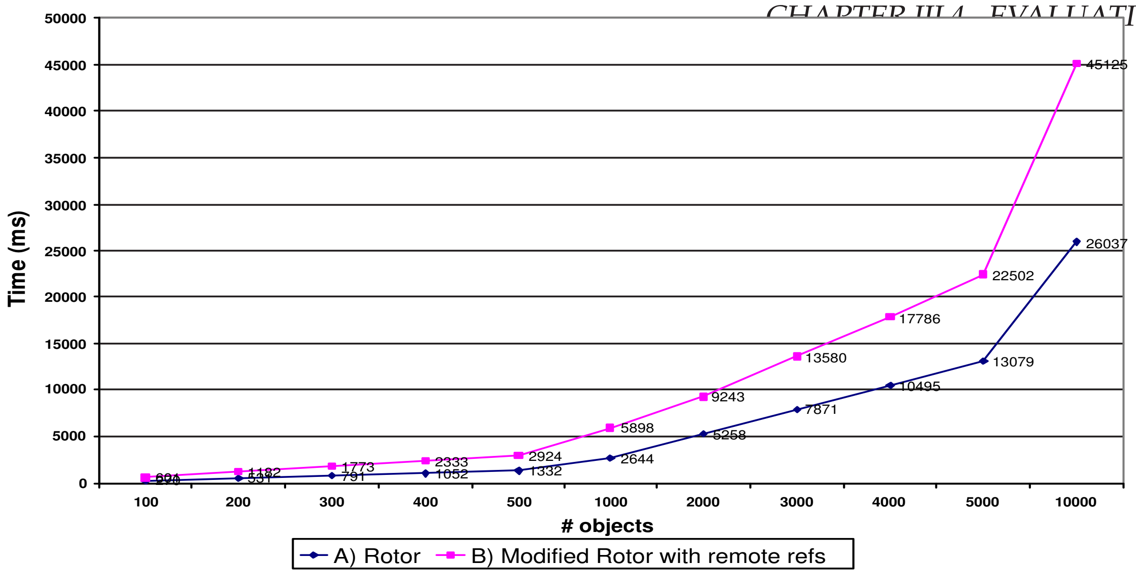
Figure III.4.1: Performance overhead due to acyclic DGC safety rules (times in ms)

III.4.1.2 Snapshot Creation

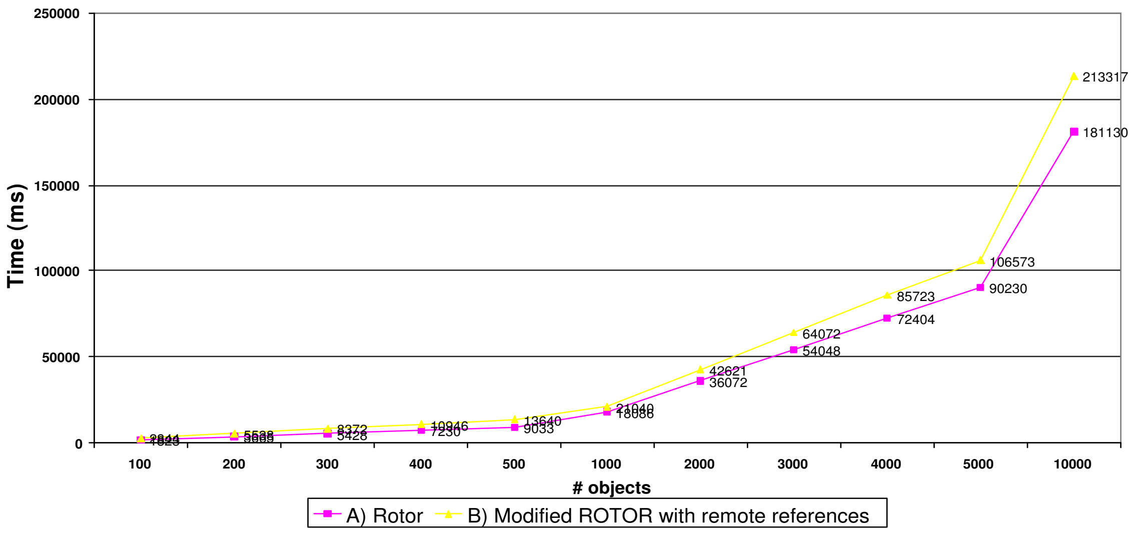
Snapshot creation needs not be performed frequently as it is only required for distributed cycle detection. Nonetheless, the performance of snapshot creation (by means of serialization of the object graph) was evaluated using three different tests: **1)** binary serialization in Rotor, **2)** SOAP-based serialization in Rotor (using XML), and **3)** binary serialization in the commercial version of .Net. The results are depicted in Figure III.4.2.

As a micro-benchmark, we used two test-graphs: **A)** graphs with an increasing number (100 to 10000) of linked *dummy* objects, i.e., just holding a reference to the next object in the list, and **B)** the same graphs, with every object containing an additional inter-process reference (thus involving the addition of at most 10000 stubs).

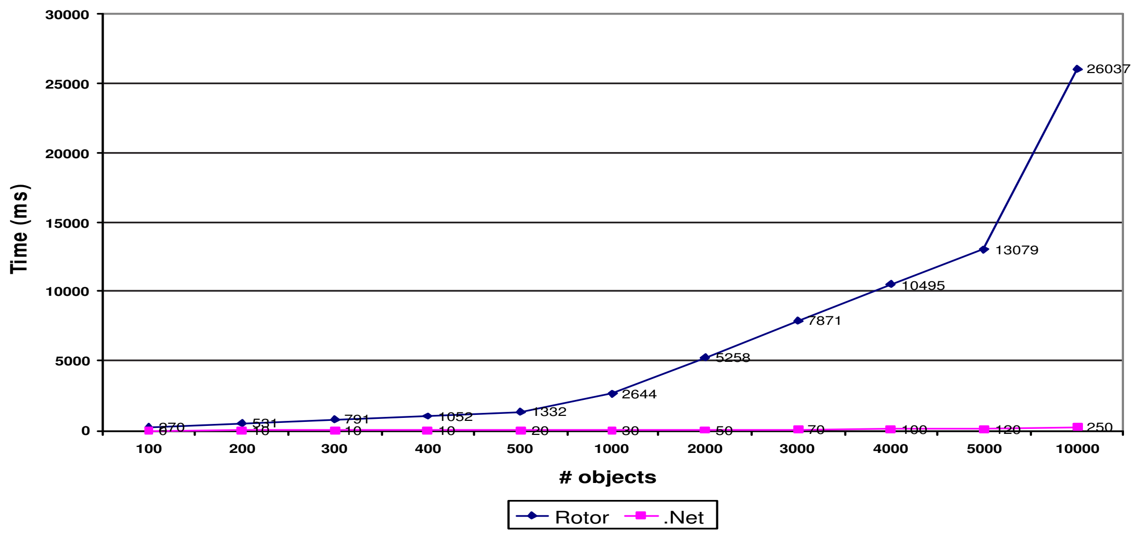
Test-graphs B portray a very conservative scenario, concerning the number of outgoing inter-process references: each object holds a single remote reference. In normal circumstances, the number of remote references in a process is several orders of magnitude lower than the number of local references. The results on Rotor (tests 1 and 2) revealed a significant performance impact, while test 3, on a more realistic scenario (.Net CLR), provided better results.



a. Test 1) Binary Serialization in Rotor: A) local references only, and B) custom-serialization of proxy objects for remote references.



b. Test 2) SOAP-based Serialization in Rotor: A) local references only, and B) custom-serialization of proxy objects for remote references.



c. Test 3) Comparison of binary serialization in Rotor and .Net commercial version.

Figure III.4.2: Performance results of Snapshot Creation.

Binary Serialization in Rotor (Test 1): Rotor serialization, using the native binary format (see Figure III.4.2-a), takes at most 26037 ms, for test-graphs A. To serialize test-graphs B, it takes 45125 ms, roughly, 73% slower. This is due to the extra work in serializing one additional inter-process reference, for each object in the graph (i.e., 10000 inter-process references).

Nevertheless, the modifications applied to Rotor enable serializing an inter-process remote reference faster than serializing an additional dummy object. They are serialized as an additional field containing the 64-bit objectID of the remote object. Therefore, the impact of serializing GC structures is lower than that of objects.

SOAP-based Serialization in Rotor (Test 2): Snapshot creation using SOAP-based serialization (see Figure III.4.2-b) is a very time consuming task in Rotor. Serializing 10000 objects in test-graph A takes more than 200 seconds, which is somewhat dismaying, considering the number of objects, and the fact that they contain no extra data besides references.

The performance impact w.r.t. test-graph B, thus including serialization times of inter-process references, is roughly, 13% slower, when 10000 objects are serialized. This is due to serializing one extra inter-process reference contained in each object. Performance penalties have lower relative impact in SOAP-based serialization because of the higher inefficiency of this serialization mechanism, w.r.t. binary serialization.

Comparison of Rotor and .Net commercial version (Test 3): The performance results w.r.t. Rotor, depicted in Figures III.4.2-a and -b, are rather un-encouraging. Nonetheless, this is not critical since it is only required for distributed cycle detection, and therefore, performed infrequently.

These results are a direct consequence of the very inefficient serialization code included in Rotor. This code is regarded as serving purely demonstrational purposes. We believe this to be intentional as Microsoft considers several aspects of the .Net CLR (Common Language Runtime) as commercial product critical code. Namely, the serialization and LGC code shipped with Rotor originates from a different code base, w.r.t. .Net CLR. This is specially significative regarding SOAP-based serialization.

To illustrate this situation, we repeated serialization of test-graphs A, this time running with production-level .Net serialization code, whose results are depicted in Figure III.4.2-c. Serialization times are, roughly, 100 times faster, thus encouraging. It takes on average 250ms, which imposes significantly shorter pause times, reaching one full second with 40000 objects, thus growing linearly.

W.r.t. test-graphs B, remote references are implemented as OBIWAN proxy-out objects. Since these are user-level objects (as all objects in OBIWAN), no optimization is possible. Therefore, in the worst-case scenario portrayed in test-graphs B, times double (i.e. 500 milliseconds). Nonetheless, this needs to be performed only sporadically.

III.4.1.3 Snapshot Compression

To evaluate the performance of snapshot compression, we used a synthetic benchmark based on data taken from previous work on memory behavior in the Java virtual machine. We will address only one of the algorithms, Algebra-based Cycle Detection, but a similar approach can be used for the others. In the case addressed, the algorithm requires reachability information to be stored both in scions and stubs. The quantities involved, and the calculations performed, to estimate the size of a compressed snapshot, and consequent compression ratio, are described next. Results would be better in DGC-Consistent Cuts and DCC-RM, that only require *one-way* reachability information.

1. $NObj$: Number of objects in memory.
2. $NScions$: number of scions (i.e., distinct incoming inter-process references targeting objects in the process).
3. $NStubs$: number of stubs (i.e., distinct objects located in other processes, targeted by inter-process references, contained in objects of the process).
4. $GraphSize = \sum_i^{NObj} size(obj_i)$
5. $ScionSetSize = \sum^{NScions} size(scion)$
6. $StubSetSize = \sum^{NStubs} size(stub)$
7. $size(ScionReach) = size(scion) + \lceil (NStubs + 1)/8 \rceil$ *(since 1 byte = 8 bits)*
8. $size(StubReach) = size(stub) + \lceil (NScions + 1)/8 \rceil$
9. $ScionReachSetSize = \overbrace{\sum^{NScions} [size(scion) + \lceil (NStubs + 1)/8 \rceil]}$
10. $StubReachSetSize = \overbrace{\sum^{NStubs} [size(stub) + \lceil (NScions + 1)/8 \rceil]}$
11. $SnapshotSize = GraphSize \quad + \quad ScionSetSize \quad + \quad StubSetSize$
    $SnapshotSize = \overbrace{\sum_i^{NObj} size(obj_i)} + \overbrace{\sum^{NScions} size(scion)} + \overbrace{\sum^{NStubs} size(stub)}$
12. $CompressedSize = ScionReachSetSize \quad + \quad StubReachSetSize$
    $CompressedSize =$
    $\overbrace{\sum^{NScions} [size(scion) + \lceil (NStubs + 1)/8 \rceil]} + \overbrace{\sum^{NStubs} [size(stub) + \lceil (NScions + 1)/8 \rceil]}$
13. $CompressionRatio = \frac{SnapshotSize}{CompressedSize}$

The memory behavior of Java programs has been studied in the literature (Dieckmann and Holzle 1999; Kim and Hsu 2000; Bacon et al. 2002; Lo et al. 2002), using different (and some times optimized) implementations of the Java virtual machine. In these tests, the average size of the Java heap is 150 MB. Average object size is small in Java (and expectably in .Net as well) as the previous works demonstrate, averaging 30 bytes or less.

However, previous work does not consider character, byte, and reference arrays, as private object data. They only consider the cost of storing primitive data and references to other objects. If these indirect costs are attributed to object instances, then the average object size (for most cases) will vary between 36 and 236 bytes (Lo et al. 2002). Therefore, both 64 and 128 bytes are useful and practical estimations for average object size, occupying respectively, 16 and 32 memory words, 32-bit wide.

As typical heaps in the study have 150 MB in size, that would amount to almost 2.5 million objects of 64 bytes. We *assume* that the number of objects referenced remotely, in large heaps, will be in the order of thousands; we will consider two boundary scenarios to the ratio between local and remote objects: between 100/1 and 2000/1.

To the best of our knowledge, there are no available measurements w.r.t. the actual density of remote references in application graphs, i.e., the fraction of objects involved in remote references when compared to total number of objects. The assumption employed is based on other assumptions from several sources: i) books on Enterprise Java Beans (Monson-Haefel 2000), other discussion *fora* related to EJB (Java Ranch 2001), RMI online documentation (Sun Microsystems 2004), and RMI debug information available at (Sun Microsystems 2001). All these sources suggest that medium-sized servers contain thousands of objects targeted by remote references.

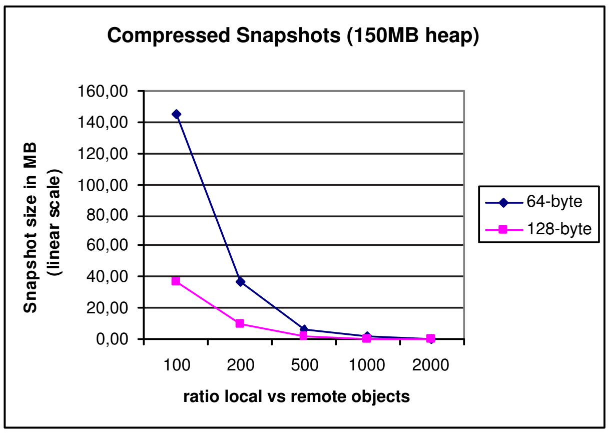
To estimate the size occupied by scions and stubs, consider that a non-optimized representation of a stub or scion, without reachability information, requires at most:

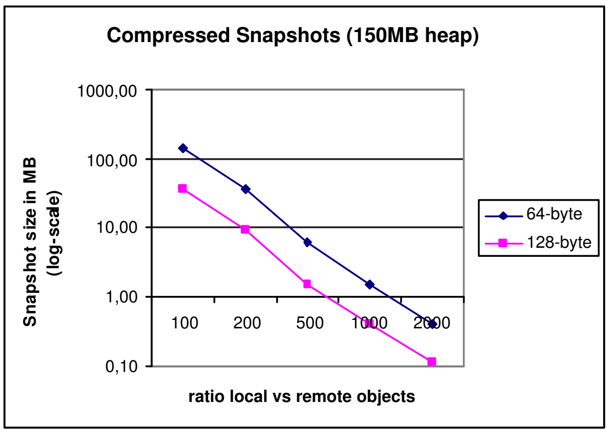
- two 32-bit words for object header (classID, lock, hashCode, etc.).
- one word for IP address.
- one word for object ID, within the process.
- one word for invocation counter/time-stamp.

In the compressed snapshot, an additional reachability bit-map is required, whose maximum size in words, is $\frac{max(NStubs,NScions)}{32}$. Thus, the base size of scions and stubs is 20 bytes, with added reachability information, that is dependent on the number of stubs and scions.

We now present the calculation for a synthetic test-case: a 150 MB heap, with average object size of 64 bytes, and a ratio of local to remote objects of 1000 to 1. We assume an equal number of scions and stubs. The results are the following:

1. $NObj = 2457600$
2. $NScions = 2458$
3. $NStubs = 2458$
4. $GraphSize = 157286400\ bytes$
5. $ScionSetSize = 49160\ bytes$, with scions and stubs occupying 20 bytes each.





Size of Compressed Snapshots (MB)					
Ratio of Local vs Remote objects	100	200	500	1000	2000
64-byte objects	144.94 MB	36.47 MB	5.95 MB	1.53 MB	0.41 MB
128-byte objects	36.47 MB	9.23 MB	1.53 MB	0.41 MB	0.11 MB

Compression Ratio					
Ratio of Local vs Remote objects	100	200	500	1000	2000
64-byte objects	1.04	4.12	25.21	97.90	367.99
128-byte objects	4.12	16.25	97.90	367.99	1321.27

Figure III.4.3: Results of Snapshot Compression for synthetic 150 MB heaps.

6. $StubSetSize = 49160\ bytes$
7. $size(ScionReach) = 20 + \lceil (2458 + 1)/8 \rceil = 20 + 307 = 327\ bytes$
8. $size(StubReach) = 20 + \lceil (2458 + 1)/8 \rceil = 20 + 307 = 327\ bytes$
9. $ScionReachSetSize = 803766 = 785\ KB$
10. $StubReachSetSize = 803766 = 785\ KB$
11. $SnapshotSize = 157286400 + 49160 + 49160 = 157384720\ bytes = 150.09\ MB$
12. $CompressedSize = 803766 + 803766 = 1607532\ bytes = 1.53\ MB$
13. $CompressionRatio = \frac{157384720}{1607532} = 97.90\ times$

Following the previous example, Figure III.4.3 presents a study of how the size of compressed snapshots varies with changes in average object size, and the ratio of local vs remote objects. The graphs show the results contained in the top table, using linear and logarithmic scales. The graphs show that the size of compressed snapshots decreases (and the compression ratio increases) geometrically, with the ratio of local vs remote objects. Doubling this ratio produces compressed snapshots with a size four times smaller. Conversely, compression ratio increases by a factor of four.

Break-even for 64-byte objects is reached with a ratio of local vs remote objects of 100. For 128-byte objects, the break-even is reached with a ratio of 50. However, note that the size of compressed snapshots also varies geometrically with average object size. For a heap with the same size, if the average object size doubles, there will be half the objects. This results in, for the same ratio among local and remote objects, a reduction of stubs and scions to half. Since the size occupied for each scion and stub also halves (approx.), the resulting snapshot will be roughly 4 times smaller.

When the number of stubs reachable from each scion (and vice-versa) is 5%, reachability bit-maps are replaced with a vector of indexes. This results in an improvement of 22,93%, thus with a compression factor of 120.36 (instead of 97.90) in the previous example.

III.4.1.4 Overhead due to Enforcement of Union Rule

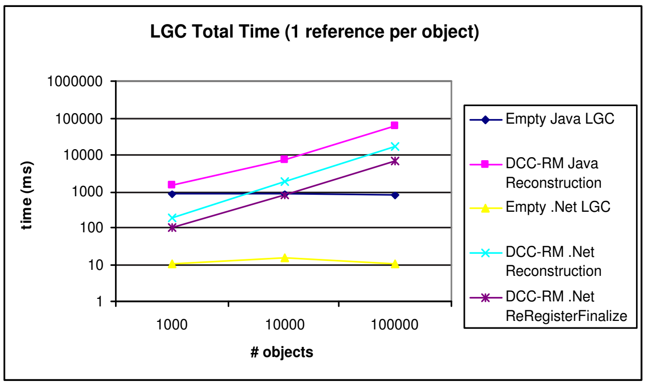
To assess the cost of enforcing the Union Rule in DCC-RM, resorting to user code inserted in object finalizer methods (see Section III.3.2.1.1), we did a series of experiments, with varying parameters. We used a Pentium 4 2.8 GHz with 512MB, equipped with .Net Framework 1.1 and Java J2SE 4.0 (version 1.4.1).

The experiments portray a worst-case scenario that provides an upper-bound to the penalties imposed. It is assumed that all objects in memory have been replicated from another process, and that they are continuously: i) being detected as unreachable locally (thus, their finalizer is executed), and ii) immediately made reachable to the mutator again (by means of a reference being imported). When this happens, the object must be handled in order that, in the future, local un-reachability will be detectable again. This is achieved using the techniques described in Section III.3.2.1.1: i) *object reconstruction*, and ii) *re-registering objects for finalization*.

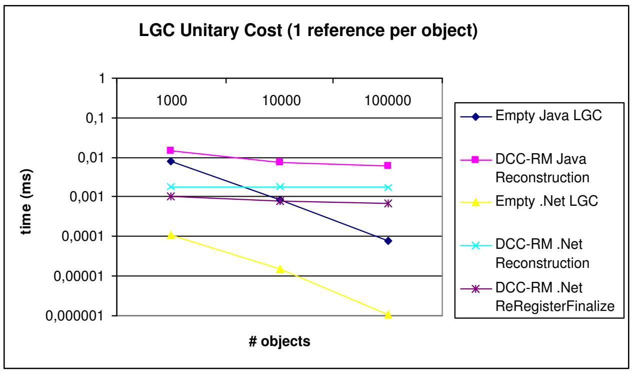
Thus, the experiments depict a worst-case scenario. In normal operation, this overhead is not imposed to all objects, nor for every execution of the LGC or object invocation. An object that is reachable locally imposes no additional overhead to LGC, attributable to the Union Rule. Similarly, an object already detected as unreachable locally, imposes no additional LGC overhead.

The overhead occurs during the transitions between local reachability and un-reachability. When an object replica becomes unreachable locally, its finalizer must be executed to resurrect the object and preserve it. When an object replica, that is unreachable locally, becomes reachable again (due to reference import), one of the two techniques described must be used. In real scenarios, these transitions happen to each object, only in a very small fraction of LGC executions. Moreover, this penalty would be completely masked if the content of the object replica is also being refreshed from another process across the network.

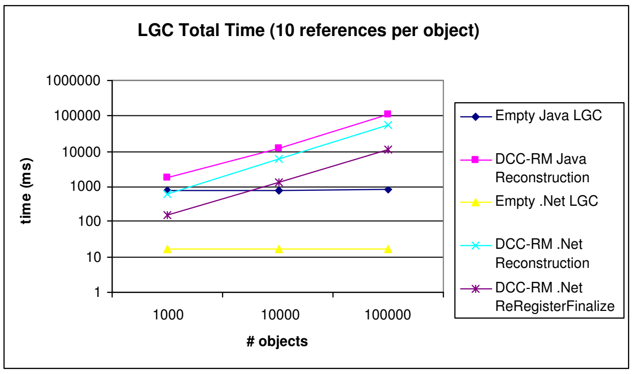
The results of the experiments are presented in Figure III.4.4, with six graphs and corresponding value tables. Each value is expressed in milliseconds and is averaged over three separate runs. In each run, the LGC of the virtual machine was explicitly executed, in loop, 100 times. This is achieved by invoking `System.gc(); System.runFinalization()` in Java, and in C#, `System.GC.Collect(); System.GC.WaitForPendingFinalizers()`. The graphs depict total execution times (a), c), and e)), and unitary cost per object (b), d), and f)), for increasing number of objects (1000,10000, and 100000), and references contained in each of them (1,10, and 25). All graphs are in logarithmic scale and contain five data series:



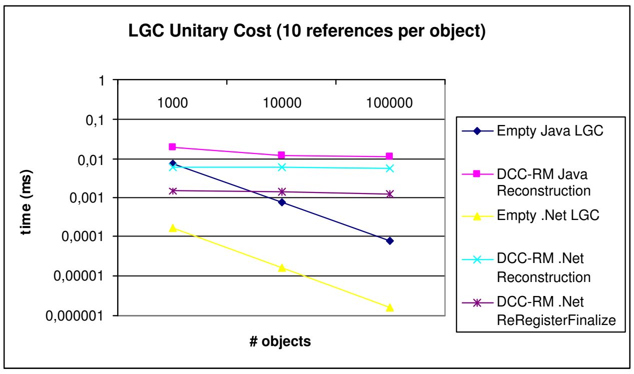
a.



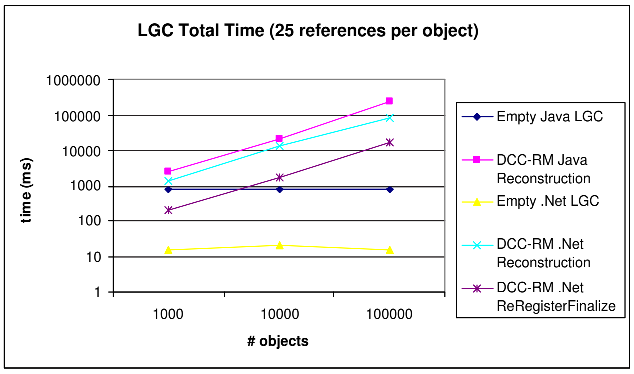
b.



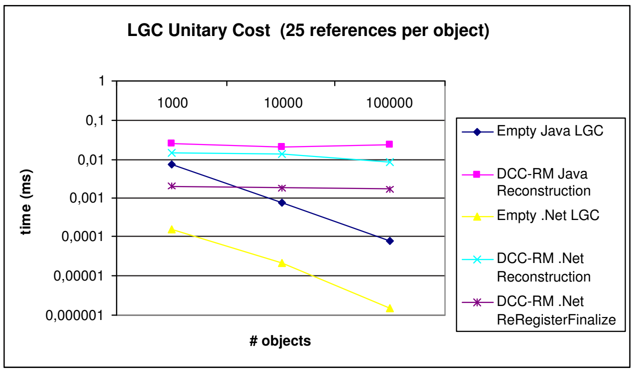
c.



d.



e.



f.

LGC Total Time	1000	10000	100000
(1 ref/obj) Empty Java LGC	807,33	802,33	786,00
Java Reconstruction	1510,00	7494,00	63198,00
Empty .Net LGC	10,67	15,33	10,67
.Net Reconstruction	182,33	1839,00	17323,33
.Net ReRegisterFinalize	98,67	791,67	6739,33

(10 ref/obj) Empty Java LGC	750,00	749,67	770,67
Java Reconstruction	1875,00	12156,33	113317,67
Empty .Net LGC	16,00	16,00	16,00
.Net Reconstruction	593,67	6020,67	57755,00
.Net ReRegisterFinalize	146,00	1328,33	11786,33

(25 ref/obj) Empty Java LGC	760,67	750,00	786,33
Java Reconstruction	2541,33	20692,67	238718,67
Empty .Net LGC	15,67	21,00	15,33
.Net Reconstruction	1448,00	13859,00	83239,67
.Net ReRegisterFinalize	203,33	1771,33	16557,00

LGC Unitary Cost	1000	10000	100000
(1 ref/obj) Empty Java LGC	8,07E-03	8,02E-04	7,86E-05
Java Reconstruction	1,51E-02	7,49E-03	6,32E-03
Empty .Net LGC	1,07E-04	1,53E-05	1,07E-06
.Net Reconstruction	1,82E-03	1,84E-03	1,73E-03
.Net ReRegisterFinalize	9,87E-04	7,87E-04	6,70E-04

(10 ref/obj) Empty Java LGC	7,50E-03	7,50E-04	7,71E-05
Java Reconstruction	1,87E-02	1,22E-02	1,13E-02
Empty .Net LGC	1,60E-04	1,60E-05	1,60E-06
.Net Reconstruction	5,94E-03	6,02E-03	5,78E-03
.Net ReRegisterFinalize	1,46E-03	1,32E-03	1,17E-03

(25 ref/obj) Empty Java LGC	7,61E-03	7,50E-04	7,86E-05
Java Reconstruction	2,54E-02	2,07E-02	2,39E-02
Empty .Net LGC	1,57E-04	2,10E-05	1,53E-06
.Net Reconstruction	1,45E-02	1,39E-02	8,32E-03
.Net ReRegisterFinalize	2,02E-03	1,75E-03	1,65E-03

Figure III.4.4: LGC overhead, due to enforcement of Union Rule, during 100 LGC executions.

- **Empty Java LGC**: Times regarding plain execution of Java LGC. In the first execution, the totality of objects are reclaimed. The remaining executions are thus empty. It provides a lower-bound on LGC execution-time in Java.
- **DCC-RM Java Reconstruction**: Times regarding Java LGC and, after each LGC execution, finalizer code that performs reconstruction of all objects, and replacement of all internal references, with proxies.
- **Empty .Net LGC**: Times regarding plain execution of LGC in .Net. It serves a purpose analogous to Empty Java LGC.
- **DCC-RM .Net Reconstruction**: Times regarding .Net LGC combined with object reconstruction in finalizer code.
- **DCC-RM .Net ReRegisterFinalize**: Times regarding .Net LGC and, after each LGC execution, finalizer code that simply re-registers the object for finalization, thus allowing the finalizer to be continuously executed, avoid additional object and proxy creation.

Empty LGC: Empty LGC takes nearly constant time, irrespective of the number of objects initially created and the number of references contained in each of them (from 1 to 25). Thus, total times for Empty LGC are approximately constant throughout graphs a), c) and e). Therefore, the unitary cost, per object, of Empty LGC decreases linearly as the number of objects increases.

The LGC in .Net performs significantly better than in Java. The use of the built-in JIT[1] in .Net cannot alone explain a speed-up of around 50 times, since Empty LGC is mostly executing VM code (that should already be assembled/binary), and not application code.

DCC-RM Java Reconstruction: Enforcing the Union Rule of DCC-RM in Java is restricted to the use of one technique: object reconstruction. Graphs a), c) and e) show that the total time used to reconstruct all objects, for the duration of 100 LGC executions, grows linearly for larger numbers of objects (10000 and 100000). Total times increase sub-linearly as the number of internal references increases (i.e., the number of necessary proxy creations to replace internal references to other replicas), doubling with a ten-fold increase in the number of references, and quadrupling with a 25-fold increase. This demonstrates that the insertion and execution of finalizer code in such a large number of objects, during so many LGC executions, does not cripple LGC performance and scales well.

Graphs b), d) and f) show that the unitary cost of each object reconstruction in Java remains constant, in each graph, for larger numbers of objects reconstructed. There is a four-fold increase in reconstruction time when the number of internal object references raises from 1 to 25. This series contains the highest values w.r.t. unitary cost of LGC.

DCC-RM in .Net: W.r.t enforcing the union Rule of DCC-RM in .Net, there are two available options, when objects previously found as unreachable locally, become reachable again to the

[1]Just-In-Time compiler.

local mutator in the process. Objects may be reconstructed as in Java, or the objects may be registered again for finalizer execution, using method `System.GC.ReRegisterForFinalize`, abbreviated as ReRegisterFinalize.

DCC-RM .Net Reconstruction: The results regarding the enforcement of the Union Rule of DCC-RM, using object reconstruction in .Net, follow the same trends as in Java. Total times for LGC increase linearly with larger number of objects, and sub-linearly with the number of internal references, in each object, that must be replaced with proxies. A ten-fold increase in the number of such references produces a maximum slowdown of 3.33 times. When the number of references increases to 25, the slowdown observed is 8.0 for 1000 objects, decreasing to 4.8, with larger graphs (100000 objects). This indicates that the code inserted in object finalizers scales well as the number of objects in the heap increases, and therefore does not disrupt LGC execution.

W.r.t. the unitary cost of enforcing the Union Rule, using object reconstruction in .Net, values are constant as the number of objects increases but, understandably, are higher for objects with larger number of internal references. Even though, this increase is less than 8.5 times when the number of references replaced, for each object, climbs from 1 to 25 (from 1.73 to 14.5 milliseconds).

The important difference w.r.t. using object reconstruction in Java and .Net is in the absolute values of the penalty imposed. Total times for object reconstruction in .Net are, globally, much smaller than in Java. The speed-up ranges from 1.75 (1000 objects, with one reference) to 8.3 times (1000 objects with 25 references). Speed-ups between 3 and 4 are common in the remaining cases. The same relation holds when unitary costs between Java and .Net are compared.

DCC-RM .Net ReRegisterFinalize: The difference between .Net and Java implementations is even greater if, in .Net, the Union Rule is enforced using multiple invocations of finalizer code for each object. The results of re-registering objects for finalization follows the same trends described for the other two analogous cases (Java Reconstruction and .Net Reconstruction).

The speed-up w.r.t. using object reconstruction in .Net, increases as the number of references in each objects grows (from 1 to 25). This is due to the fact that finalizer registration has, in theory, a fixed cost (just adding the object to a list), regardless of object size, while the cost of object reconstruction is dependent on object size and number of references.[2] Speed-ups observed range from approximately two-fold (for 1000 and 10000 objects with one internal reference) to 7.8 times (for 10000 objects with 25 references each). In the case of maximum graph and object size, the speed-up suffers a reduction but still outperforming object reconstruction in .Net by a factor of five.

Enforcement of the Union Rule of DCC-RM in .Net using ReRegisterFinalize is so much more efficient that it even beats the times, global and unitary, of Empty Java LGC, for smaller graphs and with lower density of references (1000 objects,with any number of references per

[2]Nonetheless, there is an observable increase in unitary costs when the number of internal references increases. Additionally, this unitary cost decreases slightly, as the number of objects increases, due to amortization of the base penalty due to plain LGC.

object, and 10000 objects, with one reference). This very significative effect also takes place in .Net Reconstruction, only in a smaller extent.

Summary: The results presented show that, although there is no support in existing LGC, in Java and .Net, to provide differentiated information regarding object reachability, it is feasible to enforce the Union Rule resorting to user-level code. The experiments evaluate the combined cost of the two operations required: i) detecting local un-reachability to preserve objects, and ii) ensure that when an object becomes reachable to the local mutator, again, it will be possible to detect local un-reachability again in the future (using object reconstruction and re-registration for finalization). Even in the worst-case scenario portrayed (both in frequency of the operation and absence of network communication), unitary costs are in the order of microseconds. Maximum values are of 25.4 in Java, and 14.5 for .Net, while the minimum are 6.32 for Java and 0.67 for .Net, which is actually lower than the unitary cost of Java in cases identified already.

III.4.2 Evaluation of DGC in Web-systems

This section presents a brief evaluation of the impact of adopting DGC algorithms (e.g., the ones presented in Section III.2) in current web-systems, both in terms of performance, as well as usability and integration. W.r.t. performance, we analyze the overhead introduced by DGC on web document transfer, in Section III.4.2.1. This is due to SRPs intercepting replies from web-servers in order to create and manage relevant DGC structures (e.g., scions). This is assessed by using a synthetic benchmark comprised of real files from popular web-sites. The usability of such a DGC-managed web-system, and its integration with today's web architecture (namely web-proxies and web-caching), is addressed in Section III.4.2.2.

III.4.2.1 Performance

Global performance, as perceived by users, is just marginally affected. In the case of URL-replacing mechanisms mentioned before, they are already in practice in several web sites, and users do not perceive any apparent performance degradation. The system makes use of similar techniques to parse URLs included in dynamic web content. We should stress that, in terms of performance, this is a much lighter operation that URL-replacing.

To evaluate performance penalties imposed by the use of DGC, we assessed increased latency in web-servers replies, due to processing in the SRPs. We performed several tests with two widely accessed sets of files, parsing the URLs included in them. These sets were obtained by crawling two international news sites: *bbc.co.uk* and *www.reuters.com* with a depth of four. These sets of files include both static and dynamically generated content. A Pentium 4 2.8 GHz with 512MB was used.

The distribution of files, from both sites, according to the number of URLs enclosed, is shown if Figure III.4.5. The *www.reuters.com* test-set comprised 313 files, including 57856 URLs. On average, each file included 184 URLs, with a minimum of 49 and a maximum of 637. It took, on average, 12.7 milliseconds more to serve each file, due to parsing.

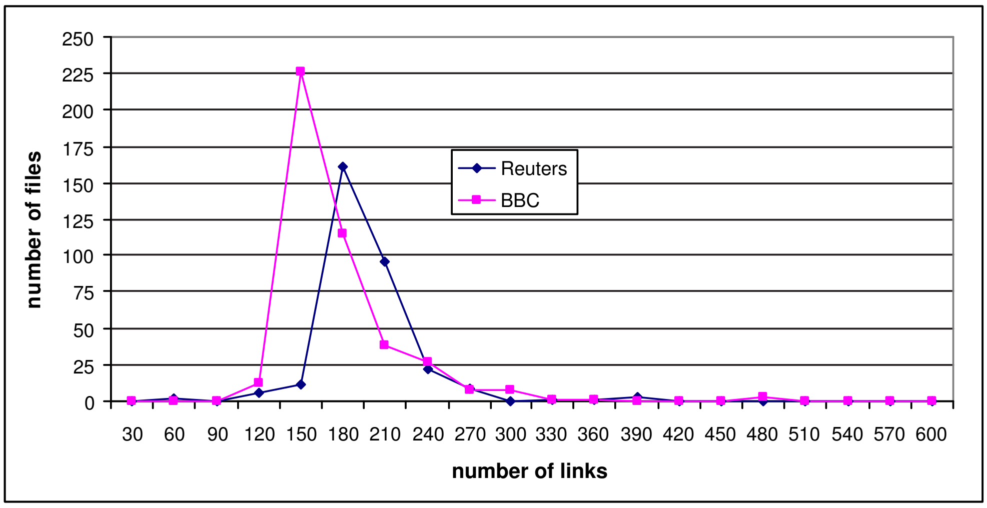


Figure III.4.5: Distribution of links per file for two sample web sites.

The *bbc.co.uk* test-set comprised 439 files, including 70401 URLs. On average, each file included 160 URLs with a minimum of 114 and a maximum of 440. On average, it took 11.8 milliseconds to parse each file. These results provide a upper-bound of performance penalty because they assume all the URLs enclosed in a document refer to a different file, and that previous information regarding URLs is either unavailable or outdated (e.g., due to document updates).

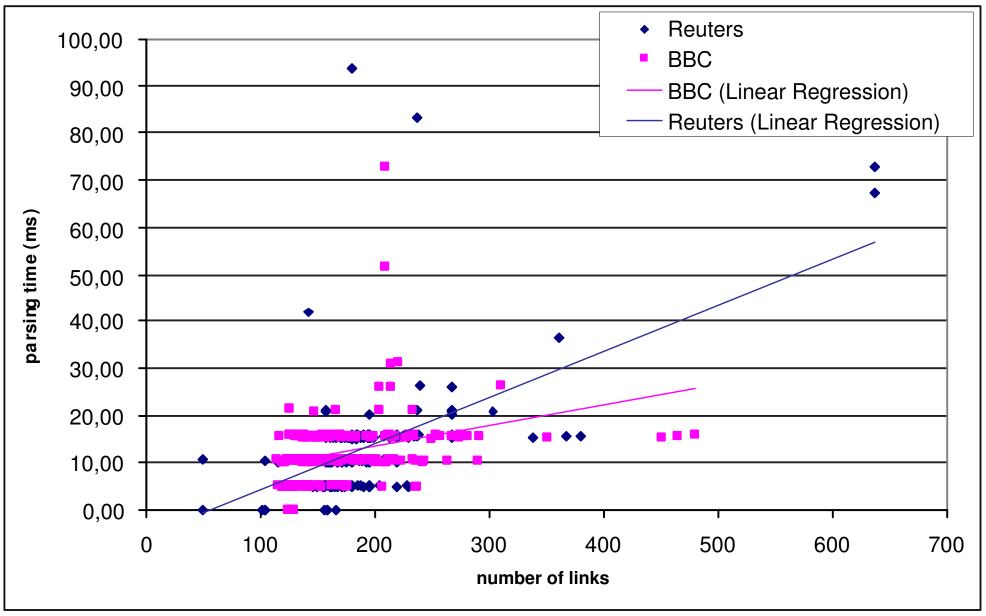


Figure III.4.6: Scattering of files based on number of links and parsing time.

Figure III.4.6 shows, for each web-site, the distribution of time spent in parsing versus the number of URLs found in each file. Linear regression allows discard of outstanding results. Differences in tendency lines reflect mainly different density of URLs in files. Broadly, files from site *bbc.co.uk* have higher density of URLs. Therefore, in this web site, a greater portion of file text consists in URLs, since the average cost of parsing each file, amortized for every URL found, is smaller.

III.4.2.2 Usability and Integration

The WWW owes a significant part of its success until now, to the fact that it allows clients and servers to be loosely coupled and different web sites to be administrated autonomously. Therefore, our system, while providing interesting properties to a set of adhering sites, does not impose total world-wide acceptance in order to function. Integration with the web can be seen from two perspectives, client and server.

Regular web clients (i.e., not connected to any extended web-proxy) can freely interact with server reverse-proxies, possibly mediated by regular proxies, to retrieve web content. However, they cannot preserve web resources or interfere with the DGC in anyway. Thus, browsing and referencing content will not prevent it from being eventually reclaimed, since these references can be regarded only as weak-references. References contained in indexers are a particular case of these weak-references.

Regular web applications in servers do need not be modified to make use of referential-integrity and DGC services. However, once a file is identified as garbage, the proxy must have some interface with the server machine to actually delete or archive the object. If proxy and server reside on the same machine, this interface can be the actual file system.

Distributed caching is widely used on the web today. It is a cost-effective way to allow more simultaneous accesses to the same web content and preserve content availability in spite of network and server failures. Caching is performed, mainly, at four levels: I) web servers, e.g., dynamically generated and periodically updated content, II) proxies of large internet service providers, III) proxies of organizations and local area networks (several of these can be chained), and IV) the very machine running the browser. Due to this structure, the web relies on caching mechanisms that have an inherent hierarchical nature. This can be exploited to improve performance (Chiang et al. 1999).

Hosts performing levels II and III caching are transparent, as far as the system is concerned. They can be implemented in various ways provided they fulfill the HTTP protocol. To perform level I, we propose a solution based on analysis of dynamic content. Server replies are intercepted by the SRPs and URLs contained in them are parsed, before the content is served to requesting clients and proxies. This is not intrusive neither for applications nor for users. Similar techniques have already been applied, as part of marketing-oriented mechanisms (e.g., *bloofusion.com*, custom web-server modules (Apache Software Foundation 1997)). These convert dynamic URLs into static ones, to improve site ratings in search engines such as *google.com*. They also allow web crawlers to index various results from different executions of the same dynamic page.

Dynamic web content can also be pre-fetched, i.e., cached in advance (Swaminathan and Raghavan 2000), based on user behavior identified and predicted using genetic algorithms. Results show that pre-fetching is effective mainly for files smaller than 5000 bytes. Such techniques could be combined with our system in order to handle dynamic content more efficiently while enforcing referential integrity.

III.4.3 Algorithm Comparative Evaluation

Following the discussion of the relevant properties of the three algorithms (Sections III.2.1.3, III.2.2.3, and III.2.3.6), we present a comparative evaluation with other relevant work found in the literature (already described in Chapter III.1). The three algorithms have a number of advantages in common, w.r.t. previous work, discussed next.

The proposed algorithms do not require all processes to participate in cycle detection in order to make progress, i.e. detect any cycle. Only those processes that comprise a cycle need participate in its detection. They require no global synchronization among application processes to perform cycle detection. The algorithms do not require that participating processes maintain state about ongoing cycle detections. Other work depends on this information for safety, completeness, or termination.

Cycle detection requires no interference with the operation of the LGC and acyclic DGC. Moreover, it does not impose the adoption of specific solutions for LGC or for acyclic DGC. The algorithms do not delay acyclic DGC because of cycle detection. Cycle detection does not impose a continuous burden to all processes. Since compressed snapshots are stored separately from processes, they do not require neither continuous update whenever a LGC occurs, nor synchronization with remote invocations. W.r.t. networking, bandwidth usage is limited due to the use of compressed snapshots, and all messages regarding cycle detection may be sent lazily and batched.

Interaction with the mutator is limited to when snapshots are generated. There is no need of barriers being enforced in every cycle detection and distributed invocation, in order to ensure safety. Furthermore, snapshots need not be taken frequently. They are good for any number of cycle detections, until all previously existing cycles are detected. Therefore, when cycle detection occurs, there is no interference neither with distributed invocations, nor with any other GC activity.

Finally, we emphasize that the algorithms do not require any significant modifications to pre-existing LGC and ADGC algorithms in a system. Therefore, they have been implemented in the context of widely available, standard platforms, such as Java and .Net.

These advantages improve algorithm scalability, as analyzed in Chapter III.2 and portability (that will be addressed in the next Section).

DGC-Consistent Cuts: The previous approach based on GC-consistent Cuts (Skubiszewski and Porteix 1996; Skubiszewski and Valduriez 1997) can only be applied to centralized systems; it is not distributed, and is strongly dependent on specific information provided by database synchronization mechanisms. GC-cuts in databases must hold at least one, but possible more, copies of every database page. Our DGC-cuts are limited to one copy of each object, and are subject to compression. Thus, our DGC-cuts besides handling distributed graphs, are more space-efficient.

The algorithm we propose does not impose any requirements neither on message latency, nor on synchronization among participating processes w.r.t DGC, as opposed to previous centralized approaches or based on groups. Furthermore, w.r.t. group-based approaches, the algorithm does not require processes involved in cycle detection to be explicitly enrolled in a specific group, and that they register that fact in their state. In fact, processes are even unaware of which and when cycle detection is taking place.

Algebra-based Distributed Cycle Detection: The algorithm does not require, as a particular cycle detection progresses, that comprised processes be explicitly enrolled in a specific group. It also does not require objects to migrate among processes, even if only conceptually (i.e., with message exchange but without actual content transfer), in order to change groups. It is fully de-centralized, in comparison with DGC-Consistent Cuts, and introduces the notion of an algebraic matching procedure for distributed cycle detection.

The algorithm does not require propagation of any information, w.r.t. cycle detection, by the LGC in processes, through local objects. As a matter of fact, in other de-centralized approaches to distributed cycle detection, the LGC must propagate extended marks, dependency-vectors (DDV), or additional color-bits. In fact, our algorithm is independent of the LGC executed in each process.

It does not require, for its normal operation, that processes keep state about ongoing detections. In that respect, the algorithm is completely stateless, in the sense that CDM carry all the information regarding each cycle detection. This way, processes may be involved in any number of cycle detections simultaneously, and they are safely handled. Other works depend on this information to ensure safety and termination in the presence of multiple detections involving the same process. In some cases, this information requires the exchange of additional messages to be kept consistent.

The algorithm, although based on message forwarding, does not require any special type of message to annul, or otherwise abort ongoing detections, in order to ensure safety. Invocation counters are used to detect distributed races with the mutator. When a CDM is forwarded to a process more than once (this must happen for at least one process, otherwise, it would not be a cycle), and there were distributed invocations by the mutator in between (therefore, a false cycle), invocation-counters will denounce mutator activity, in an optimistic manner.

In comparison with previous work, our algorithm, while being complete and scalable, is more flexible. In fact, it imposes fewer and lighter restrictions w.r.t. synchronization among processes, state at each process about detections in course, and intrusion with the mutator and with the LGC.

DCC-RM for OBIWAN: DCC-RM is the first viable approach to DGC completeness in replicated memory systems. Most of the DGC algorithms found in the literature, including the other two proposed in this document, are not safe in the presence of replication. They do not encompass the notion of Union Rule (recall Sections III.1.4 and III.2.3.3).

The approaches that are indeed safe, are either explicitly incomplete, w.r.t. distributed cycles of garbage comprising replicated objects, or have weak claims on completeness, due to scalability issues. These approaches demand that distributed cycles be fully replicated in a single process, in order to be detected, or impose full-stop to all processes in order to perform global sequential marking. DCC-RM imposes no such strict limit on the size of detectable cycles, thus being more scalable. Cycle detection is essentially asynchronous w.r.t. mutator, and other activities related to LGC and acyclic DGC.

DCC-RM inherits the properties and advantages of DGC-Consistent Cuts and DGC-WARM, whose optimization was also described (see Section III.2.3.3.3.1). Therefore, it is safe, complete and scalable for DGC in replicated memory systems. Furthermore, and contrary to previous works that impose the use of a specific run-time, DCC-RM can be readily deployed (e.g., included in OBIWAN) on existing commercial VMs, such as Java and .Net.

III.4.4 Analysis of Algorithm Portability

Most of the GC solutions[3] found in the literature are developed towards very specific systems, namely research prototypes, where it is assumed the DGC developer has complete control over the runtime. When applied to a widely deployed runtime (as Java and .NET), these solutions frequently require significant modifications to the underlying virtual machine.

We present a comparative evaluation of proposed and existing work w.r.t. portability. It is based on a qualitative overview of two main aspects that may hinder the adoption of a complete GC-solution to any widely adopted runtime: i) *runtime intrusion*, and ii) *coupling* between components of the GC-solution. Each of these aspects is decomposed in sub-aspects and, for each of them, we introduce a scale of approaches with *increasing degrees* of portability and/or flexibility. Some solutions may be mentioned at different degrees because of the different techniques they employ.

III.4.4.1 Runtime Intrusion

Runtime intrusion is defined as the need to deviate from an existing runtime, in order to provide it with a specific garbage collection solution. Such deviations may be caused by different GC components, and have different degrees. Naturally, the optimum degree is not requiring any intrusion at all, and this is the case when a specific solution is not explicitly mentioned.

III.4.4.1.1 Local GC

The most inflexible technique, with respect to LGC, when adopting a GC-solution is to impose an *heterodox LGC* (Hudson et al. 1997; Louboutin and Cahill 1997), substantially different from those typically included in the runtime.

A GC-solution may require the *extension of reachability encoding* of an existing LGC. This is the case in solutions that require the LGC to incorporate in object headers, more bit-colors (Lang

[3]We use the term *GC-solution* to designate the set of components and algorithms involved in performing garbage collection, both local and distributed, and their actual implementation.

et al. 1992; Rodrigues and Jones 1998) or additional marks, like time-stamps (Hughes 1985; Louboutin and Cahill 1997; Fessant 2001), distances from GC local-roots and process identifiers (Maheshwari and Liskov 1995), or reachability-maps (Maheshwari and Liskov 1997a; Liskov and Ladin 1986).

An existing GC may also be subject to *extension of operation* that is less intrusive than the previous technique, either pre-pending or appending operations to the ones already performed by the existing LGC, such as generating stub sets (Birrell et al. 1993a; Shapiro et al. 1992a), or calculating backward references (Rodriguez-Rivera and Russo 1997).

A solution may impose *direct instrumentation*, in that the existing LGC must be suspended (Rodrigues and Jones 1998; Maheshwari and Liskov 1997a) or triggered at specific moments (e.g., when coordinating with other GC components), possibly for a partial collection over a fraction of the object graph (Rodrigues and Jones 1998).

Indirect instrumentation consists in using indirect mechanisms to detect when a local garbage collection has taken place (e.g., using `finalizer` methods on a dummy object). This technique is portable since it only resorts to user code.

DGC-Consistent Cuts (Veiga and Ferreira 2003a) and Algebra-based DCD (Veiga and Ferreira 2005a) fall under the category *extension of operation*, since these solutions were developed with the goal of extending the Rotor virtual-machine. DCC-RM (Veiga and Ferreira 2005b) and AOP-DGC (Pereira et al. 2006) require *indirect instrumentation*, since they only resort to user code for interaction with the LGC.

III.4.4.1.2 Acyclic DGC

The most inflexible technique to implement a distributed garbage collector is to *modify the communication protocol*, or impose the use of a specific one provided by a non-standard system (Hughes 1985; Shapiro et al. 1992a; Bishop 1977; Hudson et al. 1997; Vestal 1987; Louboutin and Cahill 1997; Fessant 2001; Lang et al. 1992; Rodrigues and Jones 1998) such as Thor (Maheshwari and Liskov 1995; Maheshwari and Liskov 1997a; Liskov and Ladin 1986).

Alternatively, intrusion may be confined to *modifying remoting mechanisms* and its code (Fessant 2001). If it is possible and allowed, DGC may be implemented resorting to *interception of library loading* performed by the dynamic linker, either by extending or overriding the functionality of components regarding communication and remote method invocation, without modifying code (Rodriguez-Rivera and Russo 1997).

Portable techniques include *extended communication mechanism*, resorting to extensions allowed by the runtime, such as custom sockets.

Finally, even non-intrusive extensions may be independent of the communication protocol and restricted to *extended remoting mechanisms*, such as sink chain extensions.

The acyclic DGC in DGC-Consistent Cuts and Algebra-based DCD is implemented by *modifying remoting mechanisms* in Rotor virtual machine. The AOP-DGC solution makes use of *extended remoting mechanisms*: AOP support in .Net. DCC-RM places DGC code in OBIWAN Memory Management module (user library), and proxies (automatically generated user-code). Thus, it leverages existing remoting mechanisms, requiring no changes to .Net Remoting or Java RMI.

III.4.4.1.3 Cycle Detection

Some solutions, depending on the adopted algorithm(s) may require additional *direct intrusion* in the runtime, for the purpose of cycle detection, without allowing intrusive operations to be delayed (thus, they become disruptive). Examples include suspending the *mutator* (application) while performing bit-color propagation (Rodrigues and Jones 1998), and applying barriers to inter-space invocations when back-tracing information is being calculated (Maheshwari and Liskov 1997a).

In general, most solutions also require information regarding the GC local-roots of each space, in order to differentiate objects targeted by local references, or just by inter-space references. This may be achieved by modifying the LGC, or indirectly via hints provided by the programmer. This is required because existing runtimes, neither inform about different levels of reachability, nor provide reflection services with information about stack variables. This is the approach used in all the GC solutions presented: DGC-Consistent Cuts, Algebra-based DCD, DCC-RM, and also used in AOP-DGC.

III.4.4.2 Coupling of GC components

Coupling is defined as the degree of interdependency among different GC components (namely LGC, acyclic DGC, and cycle detection), in the sense that the adoption of one approach for one component, will mandate the adoption of the same or related approach to one, or both the others. In essence, this assesses how monolithic a GC approach is, or how it may be flexibly combined with others. This will determine the difficulty of deploying the algorithm when modifications to the runtime (namely its LGC) are not an option. Furthermore, this may hinder application performance and/or delay garbage reclamation since garbage of the three kinds is not created at similar rates, and thus should be addressed with specialized approaches.

III.4.4.2.1 One-size-fits-all

The most inflexible solutions are those that mandate the use of the *same algorithm*, a specific one for all three GC components (Hughes 1985; Hudson et al. 1997; Louboutin and Cahill 1997; Fessant 2001), i.e. the use of a acyclic DGC algorithm, or cycle detector, effectively mandates the use of the same algorithm for LGC purposes. Naturally, this seriously undermines the adoption of these algorithms to an existing runtime, if one of the components cannot be modified or extended.

III.4.4.2.2 LGC and Acyclic DGC

Some solutions demand strong integration of the components that perform LGC and acyclic DGC. They may require the LGC to *propagate information*, through the object graph, received by the acyclic DGC component, namely marks (Lang et al. 1992) and time-stamps (Hughes 1985; Louboutin and Cahill 1997; Fessant 2001), or otherwise provide inter-space *reachability information* of objects to the DGC (Liskov and Ladin 1986).

III.4.4.2.3 Acyclic DGC and Cycle Detection

There are solutions that, while avoiding intrusive modifications to the LGC of an existing runtime, use the *same algorithm* for acyclic and cyclic DGC (Hughes 1985; Hudson et al. 1997; Louboutin and Cahill 1997; Fessant 2001; Liskov and Ladin 1986). This is not as prejudicial as with the case of LGC, but it may prevent the use of a cycle detector if it imposes changes to an existing acyclic DGC algorithm (e.g., reference-listing) integrated in the runtime. Furthermore, using the same algorithm may delay the identification of acyclic garbage that should be performed more frequently (e.g., (Hughes 1985; Hudson et al. 1997; Louboutin and Cahill 1997)).

At an intermediate level, the DGC must be able to *cooperate* with the cycle detector, e.g., performing simulated deletions (Vestal 1987).

Other solutions use *specialized cycle detectors* that do not interfere with normal, more frequent, acyclic DGC operation, namely (Bishop 1977; Lang et al. 1992; Rodrigues and Jones 1998; Maheshwari and Liskov 1997a; Rodriguez-Rivera and Russo 1997). Specialized cycle detectors are used in DGC-Consistent Cuts, Algebra-based DCD, DCC-RM, and AOP-DGC.

III.4.4.2.4 LGC and Specialized Cycle Detection

The coupling between LGC and cycle detection, in the context of solutions that use the same algorithm for acyclic and cyclic DGC was already addressed in the second headed paragraph. With respect to solutions with specialized cycle detectors, those based on migration techniques must be able to *detach objects* from the local graph and create the appropriate inter-space references to preserve their reachability (Bishop 1977; Hudson et al. 1997; Maheshwari and Liskov 1995).

Trial deletion for cycle detection requires the LGC to provide *tentative reachability information* about the outcome of simulated deletions (Vestal 1987).

Cycle detection with group-merger (Rodrigues and Jones 1998) requires the LGC to *propagate information* throughout the object graph in a process, namely reachability bits (colors red and green) of ongoing cycle detections.

Cycle detectors that need to be informed about local root-sets, do not necessarily preclude the use of the runtime built-in LGC (Lang et al. 1992; Rodrigues and Jones 1998; Maheshwari and Liskov 1997a; Rodriguez-Rivera and Russo 1997), as is the case with DGC-Consistent Cuts, Algebra-based DCD, DCC-RM, and AOP-DGC.

Summary: In this section, we have analyzed the virtues and shortcomings of a number of the most relevant GC-solutions found in the literature, with respect to runtime intrusion and coupling among their components.

There is no optimal solution, i.e., one that does not require any modification nor extension of the runtime. Nonetheless, we believe that those with increased degrees of portability (i.e. runtime intrusion) and flexibility (i.e. component decoupling), can be deployed realistically, i.e., in existing systems.

Summary of Chapter: In this chapter, we presented the quantitative and qualitative evaluation of the DGC implementations presented in the previous chapter. The results obtained support the feasibility of the approaches followed. We presented performance results, in the context of distributed and replicated object systems, regarding the most relevant situations during GC operation: i) overhead of acyclic DGC rules, ii) snapshot creation, iii) snapshot compression, and iv) enforcement of Union Rule in DCC-RM.

The overhead due to enforcing acyclic DGC rules during reference export/import was measured in a worst-case scenario, i.e., running two processes in the same machine without resorting to network communication. Results vary from 7% to almost 21% slowdown, which would be mitigated by actual communication times.

The results regarding snapshot creation in Rotor were unencouraging, even though it is only seldom performed, due to inefficiency in Rotor serialization code, assessed by comparing it with the commercial version of .Net. Snapshot compression was evaluated using a synthetic benchmark based on estimations of: i) heap sizes, ii) average object size, and iii) density of remote references in a graph.

The overhead associated with enforcing the Union Rule was thoroughly measured both in Java and .Net, with different number of objects and number of references inside each object. They show it is feasible to enforce the Union Rule resorting to user-level code. Penalties range from less than a microsecond to 25 microseconds.

Concerning web systems, we evaluated the impact of DGC in web-server performance, and the adequacy of the integration proposed in the previous chapter, w.r.t., avoiding intrusion with existing browser, servers, and web-caching.

This chapter ended with a qualitative comparison of each of the algorithms proposed, with the relevant related work (frequently cited algorithms) in their application scenario. We also provided an analysis regarding the portability of GC solutions in a context where virtual machines have become the *de facto* execution environment for the majority of applications.

III5 Conclusion

Part III of this dissertation was dedicated to Complete Distributed Garbage Collection. We presented a comprehensive solution to address memory management for distributed and replicated object systems. We presented two novel algorithms for detection of distributed cycles of garbage occurring in distributed object systems with remote invocation. We also presented the first viable distributed garbage collection algorithm that is both safe in the presence of replication, and complete w.r.t. distributed cycles of garbage comprising replicated objects.

We presented a thorough survey of garbage collection (GC) algorithms and techniques. It started with a brief overview of GC for centralized systems (i.e., local garbage collection), and then we comprehensively addressed distributed garbage collection algorithms (DGC), applicable to distributed object systems based on remote invocation, to manage distributed graphs of objects. We addressed DGC in replicated memory systems, with special focus on issues regarding safety and interference with consistency mechanisms, which are handled by the Union Rule.

The three algorithms presented share the goals of being safe, complete, scalable, and asynchronous. They rely on the existence of components for acyclic DGC and LGC in each process, and operate by explicitly deleting DGC structures (e.g., scions) preserving distributed garbage cycles. Their design minimizes interference with existing support for LGC in virtual machines (i.e., they do not impose a particular approach to LGC), and with applications. DGC-Consistent Cuts employs a centralized approach. It uses a dedicated server, the distributed cycles detector (DCD), which is contacted by application processes. It constructs DGC-consistent cuts by combining representations of object graphs received asynchronously from application processes. Then, the algorithm performs a conservative mark-and-sweep (CMS) on the DGC-consistent cut.

Algebra-based Distributed Cycle Detection employs a de-centralized approach, based on a Cycle-Detection Algebra (CDA), in order to test whether a cycle candidate indeed belongs to cyclic garbage. Processes forward Cycle Detection Messages (CDM), containing CDA elements. Cycle detection is initiated by issuing a CDM regarding a scion targeting a suspect-object. If the CDM is forwarded, across a number of processes and in the absence of mutator activity, back to the originating process with all its dependencies resolved, then, a distributed garbage cycle has been found.

DGC-Consistent Cuts for Replicated Memory (DCC-RM), extends the notion of DGC-Consistent Cut to distributed systems with data replication, thus serving as a complement to an acyclic DGC for replicated objects, such as DGC-WARM. It creates replication-aware DGC-Consistent Cuts (i.e., DCCs-RM), by combining compressed snapshots received asynchronously from application processes. It detects cycles comprised within them, by performing a conservative mark-and-sweep (CMS), while enforcing the Union Rule during DCC-RM creation, and when performing CMS on them.

We implemented the algorithms described. DGC-Consistent Cuts and Algebra-based DCD were implemented in order to provide complete DGC for .Net Remoting, in the context of the Rotor virtual machine. Regarding complete DGC in replicated systems, we presented the implementation of DCC-RM in the context of OBIWAN. Acyclic DGC rules are implemented resorting to code in proxy objects and finalization methods. The whole implementation requires no changes to existing Java and .Net VM, resorting exclusively to middleware code. We also described the main aspects concerning the implementation of DGC algorithms for web systems, such as handling dynamically generated content and integration with existing web infrastructure (e.g., browser, servers, caching) without imposing modifications to it.

We evaluated the DGC implementations both in quantitative and qualitative terms. The results obtained w.r.t.: i) overhead of acyclic DGC rules, ii) snapshot creation, iii) snapshot compression, and iv) enforcement of Union Rule in DCC-RM, support the feasibility of the approaches followed. Concerning web systems, we evaluated the impact of DGC in web-server performance, and the adequacy of its integration, w.r.t. avoiding intrusion with existing browsers, servers, and web-caching. We offered a qualitative comparison of each of the algorithms proposed, with the relevant related work. The algorithms perform distributed cycle detection asynchronously, avoiding any distributed synchronization among the processes comprising the cycles, and without delaying acyclic DGC.

We also provided an analysis regarding the portability of the GC solutions to current virtual machines. All the algorithms can be implemented without requiring modifications to existing virtual machines. This has been demonstrated by the implementation of DCC-RM in OBIWAN. The portability of DGC-Consistent Cuts and Algebra-based DCD has also been demonstrated (resorting to support for aspect-oriented programming in .Net), although having been implemented via extension of the VM.

Finally, although we have implemented the ADGC and DCD algorithms in Rotor and OBIWAN, our solutions are rather general. It is possible to apply the same ideas and, in particular the notions of the DCG-Consistent-Cut and Cycle Detection Algebra, to other platforms supporting distributed objects, and DCC-RM to other systems supporting object replication.

IV

Adaptability in Memory Management

(this page was intentionally left blank)

You will be assimilated into the Borg Collective...will adapt to service us... – in "Star Trek", created by Gene Roddenberry

Part IV addresses adaptability of the middleware to changes in the environment. It focuses on adaptive memory management mechanisms (other than garbage collection) specially tailored for resource constrained devices, as those prevailing in mobile computing environments.

An extensible and dynamically, policy-driven, adaptable implementation of the M-OBIWAN prototype is described. An aggressive memory management mechanism, *Object-Swapping*, is presented in detail. An example scenario is described to evaluate middleware policy-configurability, extensibility, and adaptability to changing environment, application behavior, and user decisions.

This Part presents some related work concerning: i) adaptable and reflective middleware, including, but not limited to, object replication and memory management, as well as ii) other aggressive memory management techniques to reduce memory usage by applications. The Part closes with some conclusions.

(this page was intentionally left blank)

IV1 Architecture and Implementation

Due to their intrinsic nature, execution environments, in mobile and pervasive computing, suffer from great and diverse variations during application execution. These variations can either be qualitative (e.g., network connection or disconnection, specific devices like printers in device neighborhood, consistency and security constrains) or quantitative (e.g., amount of usable bandwidth, memory, power available).

Applications should be able to deal with this variability of execution environments. However, application programmers should not be forced to account for every possible scenario in their coding. This is unfeasible for two main reasons: i) it is error-prone and difficult to cover all potential situations, and ii) even if correctly performed, it is highly inefficient w.r.t. productivity. Furthermore, these changes often deal with system-level issues that deviate programmers from what they are supposed to do: application-logic. Programmers should not explicitly code non-functional concerns or aspects.

Therefore, this goal can only be achieved through automatic adaptation of applications and adaptation of the execution environment itself. Reflective and adaptive middleware aims at solving these issues by: i) mediating changes in the environment in a manner easily handled by applications, and ii) reacting to changes by reconfiguring itself (either its code, internal state, module organization) in order to effectively respond to changes.

To address these issues, we describe how the OBIWAN architecture supports the definition, enforcement and application of declarative XML-defined policies. Support for declarative policies in OBIWAN is named **PoliPer**[1] (Veiga and Ferreira 2004a). It allows OBIWAN to be extensible, unifying the management of several runtime aspects in mobile and pervasive environments, components and services. Adaptability in OBIWAN is achieved by the declarative configuration of its modules and parameters, and their modification according to changes in the environment that are considered relevant by applications.

The remainder of this chapter presents the architecture and implementation, in OBIWAN, providing adaptability w.r.t. memory management. It describes two main aspects: i) policy management to enable the adaptability of both the middleware and applications (PoliPer), and ii) *Object-Swapping*.

[1]**Poli**cies for mobile and **Per**vasive Environments.

IV.1.1 Policy Management

The OBIWAN middleware platform is capable of providing the needed flexibility for application development and runtime adaptability, so that applications can cope with the multiple requirements and usage diversity found in mobile settings. Its architecture is depicted in Figure IV.1.1 (it is the same as Figure I.2.3). The adaptability of applications to the particular running scenario (connectivity, available memory, availability of other resources, and other expressed constrains) is enabled by Policy Management in OBIWAN, based on the policies provided by system administrators, application developers, or users.

Middleware adaptability in OBIWAN relies on the following features: i) a policy engine, with the extensible capability to support the specification and enforcement of runtime management policies; ii) an event-handling module to allow notification of application and modules, regarding situations they are interested in knowing about; iii) a context management module that is used to store and update information about the execution environment and surroundings; iv) a plug-able set of basic mechanisms supporting object replication, memory management, etc.; v) a set of pre-defined policies that control the mechanisms previously mentioned.

Policies are stored and categorized by nature. A policy engine receives events generated by OBIWAN modules and applications, evaluates policy rules and triggers events, handled by actions based on evaluation results.

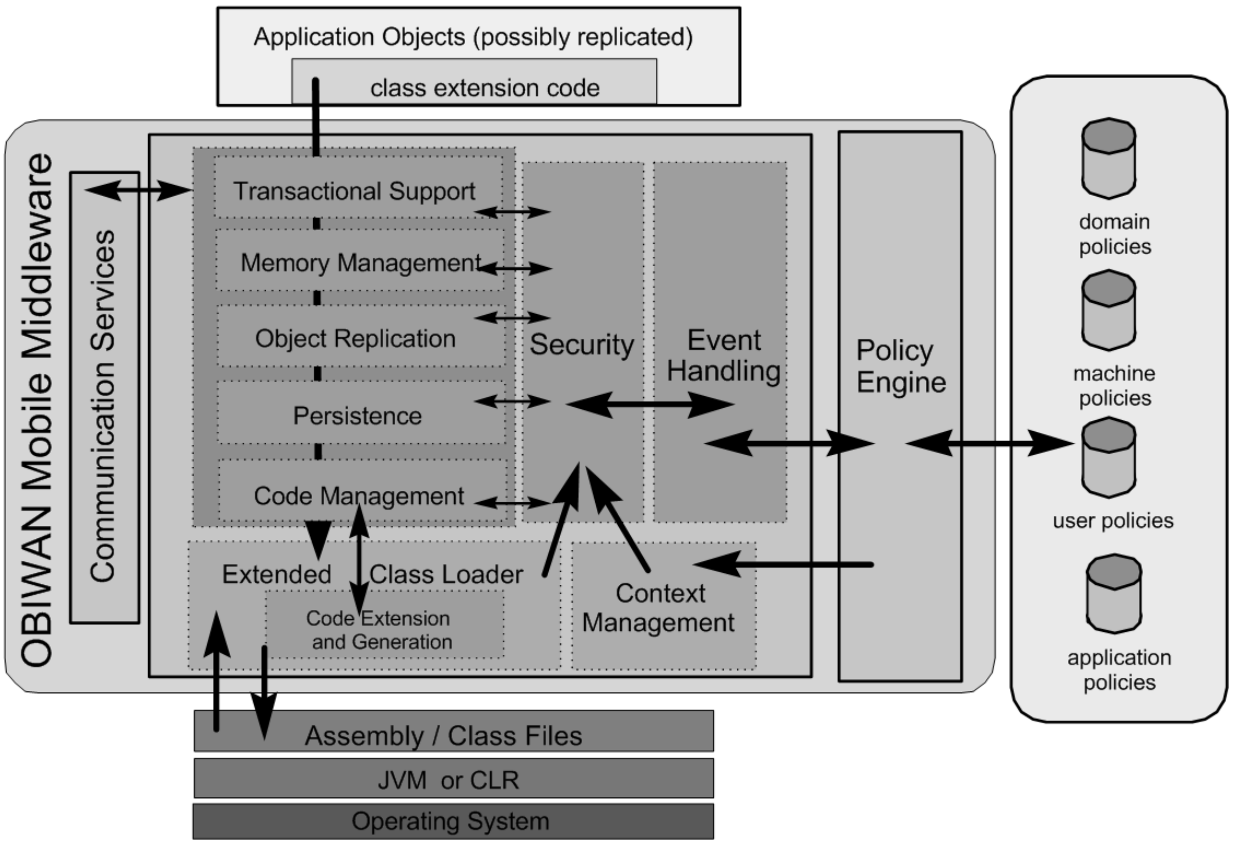


Figure IV.1.1: OBIWAN Middleware Components.

IV.1.1.1 Policy Engine

The policy engine is the main inference component that triggers or mediates responses to events occurred in the system. Apart from all the other existing (and possibly new) modules, security performs a special role, since it must be enforced by auditing or inspecting every system interaction.

The policy engine holds a variable set of policies to be enforced in the system. Policies may be encoded in XML files (an example is presented in Section IV.2.2), or created and deployed programmatically using a set of utility methods.

Policies manage, in abstract, entities. Entities are organized in an open, extensible, namespace-based hierarchy. The entity set includes resources, properties, events, user data and context information. Examples of entities are:
resource.network.connectivity.bluetooth,
property.transaction.optimistic,
event.replication.replicate-in.object.begin.

The hierarchy allows easy management of related entity-sets (e.g., *resource.network.**). Furthermore, entity-groups can be referred to with resort to regular expressions like
event.{*replication,transaction*}*.*.begin*.

A policy is a tuple: {**R**ules, **P**roperties, **E**vents, **A**ctions}.

Rules manage property changes, event triggering and handling with appropriate actions. The definition of a rule must include:

- a domain: a set of entities it relates to.
- a condition of applicability: a custom-predicate to further filter rule application.
- an event to be triggered when the rule domain and condition of applicability are met.

By decoupling the domain, the condition of applicability and the actual action-code, the system can adapt to changes in the environment and modify its own response accordingly. The difference between domain and condition of applicability stems from static versus dynamic analysis that is performed in each case.

Properties are entities with associated value (variable or not). Events are specified by the policy and registered in the event-handling module. Actions can be methods or code snippets, normally pre-defined event-handlers.

IV.1.1.2 Event Handling

Notifications to applications and to the various system modules, are performed with resort to events. Provided the necessary permissions, events can be defined either by policies (mainly) or by applications. Events can be triggered either explicitly by applications, by the system modules, or by policies when rules are evaluated. Actions performed, when events are triggered, allow OBIWAN and applications, to adapt to changes in the execution environment.

An event is a triple: {**N**ame, **S**ource, **U**serData}.

As entities, events are organized in namespaces. As an advantage, it allows event-handlers defined in policies to subscribe to specific events, as well as a whole family level of related events, e.g.:
event.replication.replicate-in.object.begin,
event.replication.replicate-in.object.end or
*event.replication.replicate-in.**

Thus, events are organized in a meaningful, yet open manner. It enables regular expression definition of events to subscribe to. This way, event names need not be fully known, or indicated exhaustively by the subscriber. Nevertheless they are intercepted and handled by the subscribers.

Event Filtering: Event jitter can be regarded as bumpiness in the continuous triggering of events, possibly with contradictory response actions. This phenomenon is frequent in execution environments (such as with mobile and pervasive ones) with frequent changes in resources and QoS[2] available to applications. These changes can trigger possibly contradictory measures and with short periods of time between them. Reacting to them too soon, too often, may hinder system performance and application behavior.

In OBIWAN, this may be avoided with resort to properties (system, user or application defined) that are evaluated both in the condition of applicability of the rule itself, and possibly updated in the action handling the corresponding triggered event. This process effectively filters events to the degree of stability desired by applications, simplifying application-logic. A straightforward example is a situation of intermittent connectivity where an application is constantly being notified that connectivity is on, and then off. If the application needs a period of stable connectivity, it can use a policy that monitors connectivity-related properties.

Policy actions hide these quick variations, and may notify the application only when there is minimum signal strength or, in alternative, when some delta time has elapsed since the last time connectivity was on.

IV.1.1.3 Context Management

The OBIWAN architecture includes a context management module. This module performs resource abstraction and manages properties whose values vary during execution. Abstraction enables representing physical machine resources as sets of primitive context properties. Examples include memory, connectivity, bandwidth available, etc. For flexibility, resources, as entities, are also namespace-organized.

The actual mappings between basic/primitive resources and resource designations is performed by the context manager. Each of these resources implies an architecture-dependent way of measuring. This heterogeneity is masked, to the rest of the system, by a low-level component

[2]Quality-of-Service.

in the context manager. Properties can be aggregated. Thus, *higher-level* properties can have their value derived from the combination of values from other, *lower-level* properties.

Situations like appearing devices, discovering remote resources or application counterparts are also handled by the context manager. The relevant properties are updated and the appropriate events are triggered. In more general terms, any change to the properties (resources, middleware state or user-defined properties) managed by the context manager can potentially trigger associated events defined by the policies loaded.

We do not specifically address adapting resources (and possibly replacing them with variants) but solely on representing them, in a flexible manner, and monitor their changing properties. Events to be triggered and actions to address them are described in policies. Appropriate policies configure context management and its events, in order to allow applications to be notified solely when these changes are stable or reach a certain threshold.

Resource management is not centralized. It is performed by the combination of security policies that monitor resource requests, and context management that registers and notifies resource shortage.

IV.1.1.4 Policy-Managed Object Replication

OBIWAN supports the specification and enforcement of policies concerning the replication of objects. Object replication is incremental and adaptive. Unless otherwise specified, it is performed transparently to applications but can also be flexibly configured by them. In particular, it allows the specification of:

- when to create a replica of an object (e.g., in imminence of disconnection).
- when to merge two or more replicas of the same object.
- the amount of objects to replicate at a given time (a cluster in OBIWAN).
- which branch of a graph should be further replicated.
- which objects should be swapped-out to a neighboring device.

The most relevant events are triggered with the replication (either in or out) of each single object:
event.replication.replicate-in.object.begin
event.replication.replicate-in.object.end, and
event.replication.replicate-out.object.begin
event.replication.replicate-out.object.end

Additionally, there are two more sets of events with coarser granularity:
event.replication..cluster.**
event.replication..graph.**

The first set handles clusters (groups of object replicated in a single time as a unit) and the second one addresses complete graph branches. Different granularity of events, triggered at different times, provide a basis for flexible management of different scenarios, e.g., latency/bandwidth tradeoffs.

IV.1.2 Object-Swapping

Mobile devices are so memory-constrained that, in some circumstances, even the memory occupied by useful reachable objects must be freed. This may occur because, at a particular instant, there are other more relevant replicas for which there is no memory available. This is more evident in resource constrained devices but also occurs in desktop systems (Chihaia and Gross 2004) even with large memory heaps.

The memory management premise of preserving live data must be enforced with a somewhat relaxed approach: there are situations where live data must be ”demoted” to accommodate for other data being replicated that is considered, at that moment, more important. Data should not be plainly discarded, but the memory occupied by it, should nevertheless be brought down. This can be achieved in a number of ways (discussed in Section IV.2.4), while guaranteeing that the data can be ”promoted” afterwards.

Our proposal consists in swapping-out such objects to other devices with more resources available, in particular, free memory. Freeing the memory occupied by useful objects is delicate. Given that such objects can be accessed by applications through navigation of the object graph, the middleware must still ensure the referential integrity, while freeing such memory.

Figure IV.1.2 depicts a prototypical scenario in which a PDA is running applications, on behalf of the user, on top of OBIWAN middleware. From time to time, the memory occupied by the object graphs of applications reaches a threshold value, possibly near the limit of the memory capacity of the device. At those moments, the OBIWAN middleware, evaluating the policies loaded, decides to swap-out a set of objects to nearby devices, if there are any. This action frees some memory while not discarding the swaped objects permanently. Later, each set of objects previously swapped-out may be fetched back from the device where it was transferred.

The devices that receive swapped objects need not have neither OBIWAN nor even a virtual machine installed. They need only be able to store and return a textual representation of the serialized objects being swapped-out. If a device is able to store more than one set of swapped objects, each set must be given a unique ID (e.g., a number, a file name). This functionality can be provided via a simple web-service since the objects are serialized using XML. Therefore, objects may be swapped to desktop and laptop PCs, other PDAs (if they have a web-server installed), or future wireless devices, with extended memory capacity, present in the room.

IV.1.2.1 Management of *Swap-Clusters*

We propose an approach, that favors portability, and thus is more suited to be deployed on a myriad of existing and future devices. It resorts exclusively to user-level code and therefore does

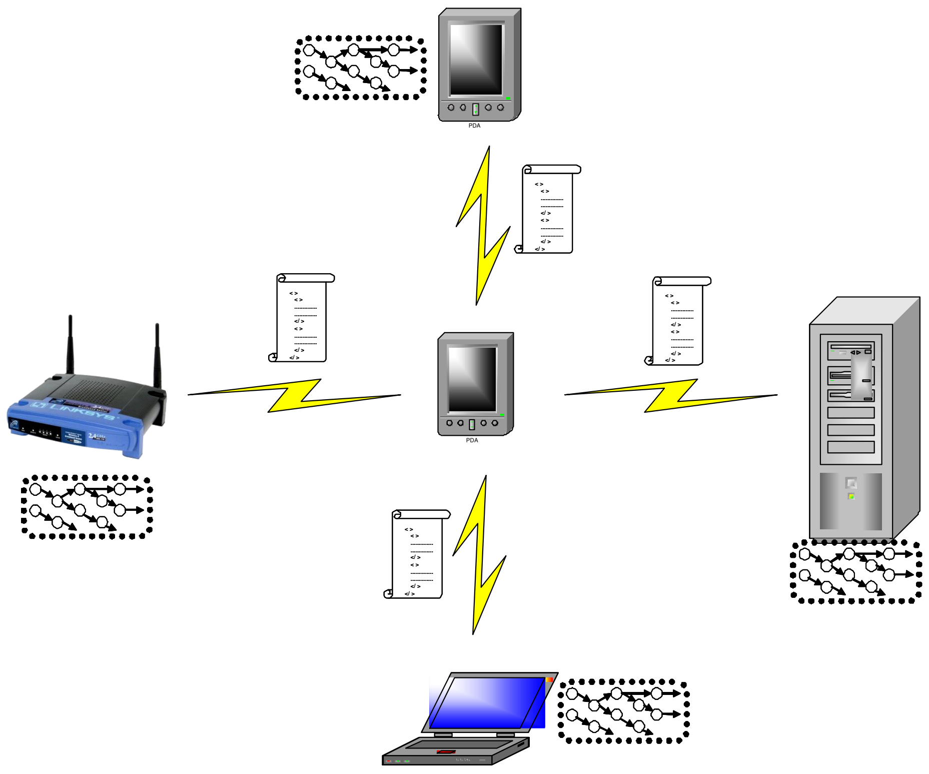


Figure IV.1.2: Object-Swapping to nearby devices.

not require modification of the underlying virtual machine, making it rather portable. It further obviates the need to manage inter-process references among individual resident and swapped-out objects. This has the negligible trade-off that every reference between objects in different swap-clusters must be mediated by a proxy.

Taking into account the management of replicas described in Chapter II.2, clusters of objects (or groups of clusters) are natural candidates to be swapped-out as they have been incrementally replicated, previously, into the mobile device as a whole. Hopefully, when one of the objects enclosed in the cluster becomes needed again, there is a high probability that the others will be as well. So, later, they will be swapped-in as a whole.

A *swap-cluster* is the basic unit of swapping. Each one contains all the objects comprised in a group of one or more object clusters, previously replicated. Swap-clusters are created by regarding a number (also adaptable) of chained (via references) object clusters as a single macro-object, i.e. a single swap-unit. For every reference linking two different swap-clusters, proxy-out replacement (that takes place when objects are replicated) is not performed as usual.

The proxy-out object is replaced with the corresponding object replica, but another type of proxy is also created. We call it a *swap-cluster-proxy* to distinguish it from proxy-out objects that once replaced, are discarded. The *swap-cluster-proxy*, in turn, holds a reference to the newly replicated object. This way, the reference returned by the middleware to application code does not target an object replica, but a swap-cluster-proxy instead. Thus, for objects belonging to different swap-clusters, a proxy always remains in the way.

Figure IV.1.3 depicts a situation in which the object graph of a process ($P1$) is divided in

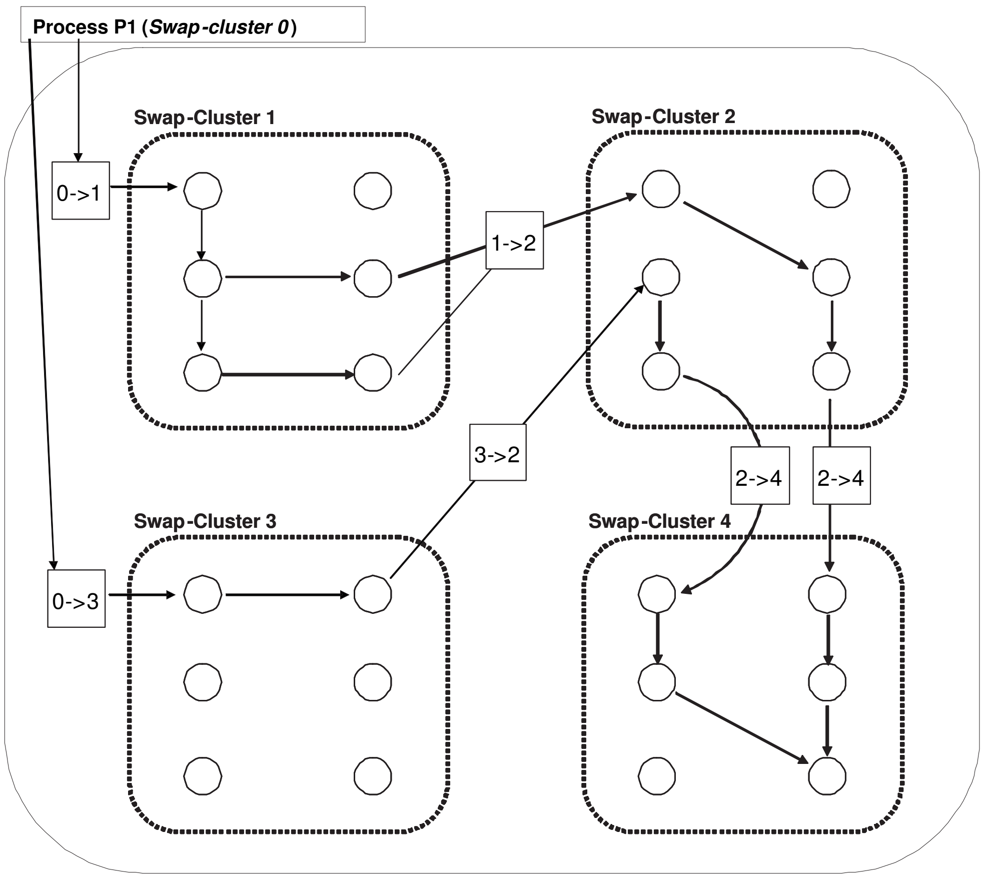


Figure IV.1.3: Object graph of a process comprising four swap-clusters.

four swap-clusters: $swap - cluster - 1$ to $swap - cluster - 4$. Global variables (i.e., static fields), and variables defined in static methods, are regarded as belonging to a special swap-cluster, $swap - cluster - 0$. $Swap - cluster - 1$ and $swap - cluster - 3$ are reachable directly from the variables of the application. Both of them contain objects that reference other objects contained in $swap - cluster - 2$. When there are multiple references to the same object, across the same pair of swap-clusters, only a swap-cluster-proxy is required. This is the case in $swap - cluster - 1$. Objects in $swap - cluster - 4$ are only referenced from objects in $swap - cluster - 2$. Each swap-cluster may contain any number of objects since the depth of incremental replication may vary over time. The only restriction is that a swap-cluster boundary must be placed only when a proxy-out is replaced.

There is a performance penalty imposed by this approach, since there are extra invocations, due to the indirection maintained in swap-cluster-proxies. Nonetheless, this only happens when an swap-cluster boundary is crossed. In favorable scenarios, they are only required with frequency inverse to the average number of objects per swap-cluster (e.g., 1/20, 1/50, 1/100), which could render them negligible.

Middleware code in swap-cluster-proxies also monitors reference-passing across swap-cluster boundaries (analogously to monitoring of inter-process references), and creates/reuses the appropriate swap-cluster-proxies. The middleware keeps track, for each swap-cluster, of swap-cluster-proxies regarding it, and their usage. This provides information about in-

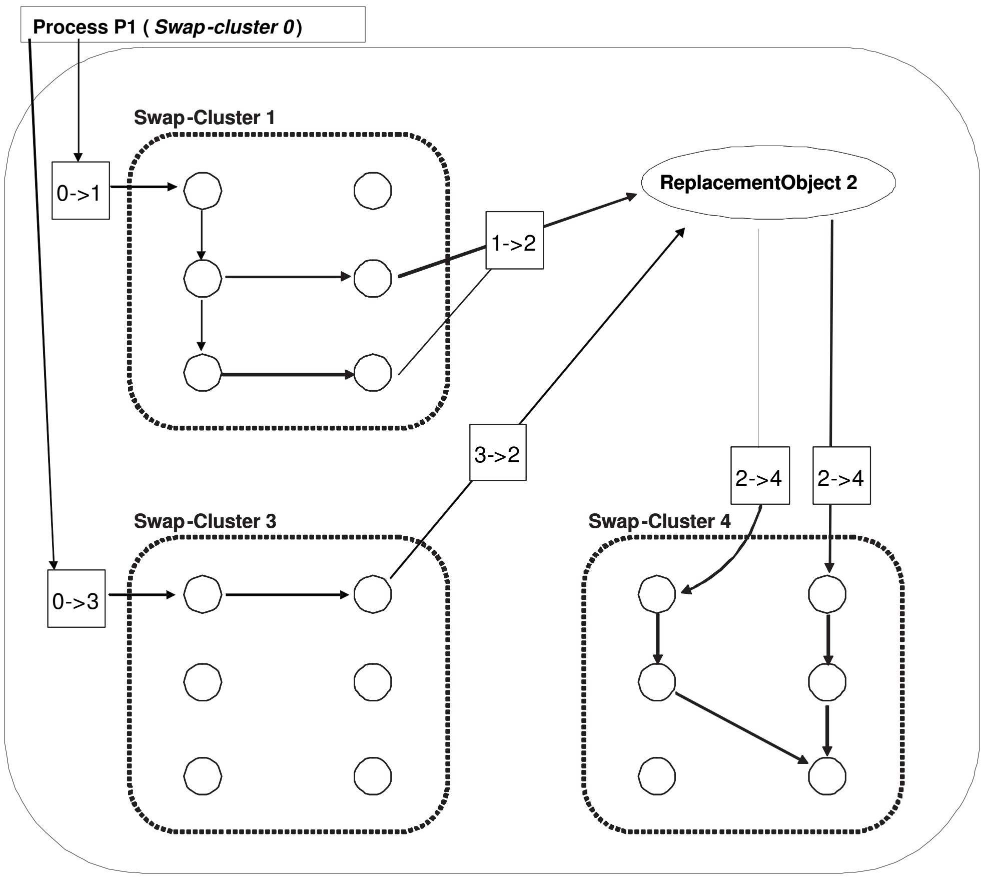


Figure IV.1.4: Object graph of a process after swapping-out of $swap - cluster$ 2.

bound/outbound references from/to other swap-clusters, and basic data w.r.t. recency and frequency, as these boundaries are transversed by the application mutator.

IV.1.2.2 Swap-Cluster Swapping-Out

When needed (e.g., for shortage of memory), the middleware may *detach* the objects belonging to a specific swap-cluster from the application graph, while maintaining correctness. This process is described by the following example depicted in Figure IV.1.4. It portrays the resulting situation after detachment of a swap-cluster, in this case, $swap - cluster$ 2.

A replacement-object for a swap-cluster (i.e., $ReplacementObject - 2$ which is simply an array of references) is created and filled with references to every swap-cluster-proxy referenced by $swap-cluster-2$. Then, every swap-cluster referencing objects contained in $swap-cluster-2$ will be made to reference $ReplacementObject - 2$ instead, by patching the internal references of every swap-cluster-proxy targeting objects in $swap - cluster - 2$.

Once a swap-cluster (e.g., $swap - cluster - 2$) is detached from the object graph (though still referenced by middleware code), the enclosed objects are serialized to XML and sent to a nearby device (e.g., via Bluetooth), along with a swap-cluster ID. The receiving device needs no other infrastructure (e.g., specific VM, adhering to a specific middleware, holding application class files, etc., as opposed to other existing solutions) other than being able to receive XML data and store it.

After a swap-cluster is swapped-out, the objects enclosed in it are completely detached from the application graph, and eligible for collection by the local GC running on the device. Thus, memory is released without destroying application graph integrity. Any swap-clusters may be swapped-out using this mechanism.

IV.1.2.3 Swap-Cluster Reload

When a replacement-object is invoked, this means that the application is trying to access an object that belongs to a swap-cluster, which was previously swapped-out. Since one of the objects enclosed in the swap-cluster becomes needed again, there is a high probability that the others will be as well. So, they are swapped-in back as a whole by demanding the same XML-data, containing wrapped objects, that was sent earlier during swap-out.

All swap-cluster-proxies targeting objects enclosed in the swap-cluster being swapped-back must be updated. To this purpose, their internal references are patched in order to target the corresponding object replicas being swapped-in. Then, the replacement-object, as it is no longer needed, becomes eligible for local reclamation. Therefore, after $swap-cluster-2$ is brought back to $P1$, the resulting object graph would be similar to the one initially shown in Figure-IV.1.3.

IV.1.2.4 Integration with GC Mechanisms

The middleware prevents dead objects, belonging to swap-clusters that later became unreachable (unusable to the application), from consuming resources while being stored on the swapping devices. Therefore, the middleware still manages DGC structures, and exchanges DGC messages, w.r.t. to those objects.

However, the reachability of a swap-cluster must be considered as a whole. This entails that when a replacement-object, standing in for a swap-cluster that has been swapped-out, becomes unreachable locally, *Unreachable* messages must be sent on behalf of all object replicas enclosed in it. Conversely, the replacement-object (and its associated swap-cluster) can only be considered as unreachable globally when *Reclaim* messages have been received on behalf of all its enclosed objects. Note that in this case, while *Unreachable* messages are sent in group, corresponding *Reclaim* messages are received one at a time (i.e., only when all other replicas of each object are found unreachable).

When ultimately deemed as unreachable globally, a swap-cluster may be dropped from the swapping node, or set-aside if their content is still required for other purposes (consistency, reconciliation, versioning, etc.). The replacement-object is simply reclaimed by the LGC.

The integration with GC mechanisms just described does not constitute a DGC infrastructure covering swapping devices. There are no explicit references among the objects residing in devices running applications, and those serialized in swapping devices. All the decisions are made locally to the application device (no reference-listing, no stub-scion pairs, etc.). The swapping device is instructed just to store, return, or drop XML-data.

IV.1.3 Implementation

Adaptability and Object-Swapping in the OBIWAN architecture have been implemented on top of the M-OBIWAN prototype, extending it (Veiga and Ferreira 2004a). It runs on .Net (for desktop and laptop nodes) and .Net Compact Framework (for SmartPhone and PocketPC). The primary programming language used is C#. Policies are coded in XML. Desktop machines have Internet Information Services installed, and a small footprint mobile web-server (Nicoloudis and Pratistha 2003) is used in PocketPC.

IV.1.3.1 Adaptability

Policies, rules, events, and properties are stored in specialized hash-tables, indexed by full-name and their name-space components, to speed-up subscriptions using regular-expressions. Policies coded in XML files are loaded by a custom XML parser coded in C#.[3] Name-spaces are identified using support for regular-expressions included in .Net (`System.Text.RegularExpressions`).

Properties in the Context Manager may be associated either with an object that holds their actual value, or with a delegate thus being procedural. In this case, the value of the property, when inspected, is the value returned by the specified method. This allows properties to invoke code that reads lower-level sensors, or to build combined higher-level properties. Examples of procedural properties[4] include: i) determining available memory, that relies on GC services, and ii) testing connectivity, that checks a variable periodically updated, by a thread that tests DNS address resolution.

The support for adaptability in OBIWAN relies on previous implementation work in OBIWAN. It requires only that the appropriate events, regarding object replication, be triggered. This includes code in proxy-out and proxy-in objects, replication management (i.e., when creating `OBIRep` entries). These are very small extensions that are automated in code generation.

Based on this fundamental support, the Policy Engine, Event Handling and Context Management modules are able to orchestrate and monitor object replication, in order to allow for extended behaviors, such as DGC and Object-Swapping.

IV.1.3.2 Object-Swapping

Code for swap-cluster-proxies is automatically generated by `obicomp`. It generates a specific class of swap-cluster-proxy, for each type defined by the application (analogous to generation of proxy-out and proxy-in objects described in Part II).

Thus, the class for each type of swap-cluster-proxy implements two interfaces: i) the `ISwapClusterProxy` interface for common methods such as `patch` and `detach`, and ii) the interface containing the public methods of the type (e.g., interface IA).

[3]This parser is based on code provided by .Net Framework Samples.

[4]These implementations rely solely on the .Net Framework. Obviously, other alternatives could be more efficient yet less portable.

The code generated for swap-cluster-proxies implements all methods of the application interface (e.g., IA), with a similar code excerpt that verifies references being passed as parameters and return values, while also relying on invoking the actual object replica it refers to. With every referenced intercepted, this code verifies whether it is necessary to: i) create another swap-cluster-proxy to wrap a reference from/to another cluster, ii) patch an existing swap-cluster-proxy that is being handed to/from another swap-cluster, iii) dismantle a swap-cluster-proxy received but that refers to an object within the same swap-cluster. Implementation of methods belonging to interface `ISwapClusterProxy` (e.g.,`patch`, `detach`) delegate to static methods of a class (`SwapClusterUtils`) that contains behavior common to all swap-cluster-proxy types.

Object Identity: Within each swap-cluster, object identity is ensured because references to object replicas are never compared against references to swap-cluster-proxies, referring to objects in the same swap-cluster, due to rule iii) of the last paragraph.

Enforcing object identity when comparing references to objects in other swap-clusters (i.e., actually comparing references to swap-cluster-proxies) cannot rely solely on reference comparison (operator `==`). A simple example would be that of an object in $swap - cluster - X$, if referenced from two different swap-clusters, will be necessarily represented by two different swap-cluster-proxies (because they regard different source swap-clusters).

This is solved by overloading the reference comparison operator `==` in C#, for each class of swap-cluster-proxy, with a method that verifies whether the two arguments received are swap-cluster-proxies (i.e., implement interface `ISwapClusterProxy`), and actually refer to the same object.

In other languages, such as Java, that do not allow this overloading, comparisons must rely solely on method `Object.Equals`, that can be overloaded. Note that this limitation in Java is only present when using Object-Swapping and does not affect regular object replication in prototype OBIWAN.Java.

Optimizing Code for Iterations: The use of global variables, while iterating object graphs (e.g., lists) that may span several swap-clusters, causes the creation of a new swap-cluster-proxy for each object returned, and consequent discard of the swap-cluster-proxy that the variable previously pointed to. In this cases, with a little help from the programmer, this behavior can be optimized re-using always the same instance of swap-cluster-proxy (as it was indeed the actual variable).

Class `SwapClusterUtils` provides a static method (`assign`) that may be invoked with swap-cluster-proxies with source in $swap - cluster - 0$. This method updates an internal field in the swap-cluster-proxy that marks it, and changes its behavior. The next time the swap-cluster-proxy intercepts a reference to be returned, instead of creating a new swap-cluster-proxy to be returned to application code (discarding itself), it patches itself. This way, it now refers to the object being returned by the application method that was invoked. Therefore, in practice, the swap-cluster-proxy will return to application code a reference to itself (though already modified internally), that minimizes creation of swap-cluster-proxies and optimizes iterations.

Swapping Manager: The `SwappingManager` class, by policy definition, is registered as a listener of all events regarding replication of clusters of objects, by using specific methods as actions. It also triggers specific events regarding object-swapping but that are presently not listened by any other module.

It manages swapping by maintaining information regarding all swap-clusters (loaded or swapped), and all objects belonging to each one,[5] stored in hash-tables. It also contains entries for all swap-cluster-proxies w.r.t. references to/from each swap-cluster (using weak-references). When a swap-cluster-proxy becomes unreachable, its `finalizer` invokes code that eliminates entries referring to it.

Summary of Chapter: In this chapter, we presented the support in OBIWAN for middleware and application adaptability to changes in the environment. For that, we described an extensible and dynamically, policy-driven, adaptable implementation of the M-OBIWAN prototype. We also presented an aggressive memory management mechanism, *Object-Swapping*, specially tailored for resource constrained devices, prevalent in mobile computing environments.

[5]Information, regarding which swap-cluster an object belongs to, can be optimized if stored directly in `OBIRep` entries.

IV.2 Evaluation

This chapter describes the evaluation of the support in OBIWAN for adaptability, and object-swapping, described in the previous chapter. The evaluation is centered on qualitative aspects. It is based on example situations (Sections IV.2.1 and IV.2.2), motivating this kind of support, that otherwise would not be handled, if static approaches were followed. We present some relevant related work in Sections IV.2.3 and IV.2.4, and offer comparative discussion.

IV.2.1 Example Scenarios

A vast number of scenarios can be imagined to portray adaptable behavior of applications designed with adaptability in mind. The developer is in charge of determining the situations the application should respond and adapt to. Then, a set of policies should be defined. They will configure the middleware to detect those situations, evaluate conditions of applicability, trigger the appropriate events and run the corresponding actions.

Other relevant scenarios are those of dynamic adaptation, to some extent, of applications that were designed without adaptability in mind. Furthermore, it is advantageous to be able to specify how applications react to certain changes in the environment (e.g., connectivity, memory shortage), without the need to write specific code for each and every one of them. This would be redundant and difficult to extend. Thus, applications can be categorized, w.r.t. adaptability to variations in the environment, along two main axis:

- Whether adaptability was taken into consideration when the application is designed.
- Whether there is the need to write application code, specifically to account for each situation.

These aspects will be the subject of the examples provided in the current and next sections. With Policy-Management in OBIWAN, we can easily set up, for this purpose, the following example scenario. A number of applications are running on a mobile constrained device (e.g., PDA). They have already replicated some data for local disconnected use. Application code simply navigates through object graphs; it is not otherwise aware of OBIWAN (i.e., the applications running were not designed with adaptability in mind). In this example scenario, the following policies are loaded and acting/reacting as follows, with increasing priority:

- Policy P_1 determines, for each application and according to the available bandwidth, the size/depth of each replication cluster.

- Policy P_2 determines that whenever connectivity is back on, and the application has accessed a threshold fraction of the previously replicated objects, another cluster of objects should be immediately pre-fetched.

- Policy P_3 determines that when there is Bluetooth connectivity, GPRS access should not be used, for economic reasons.

- Policy P_4: Swap-out objects when low on memory and connected to home LAN (via Bluetooth).

- Policy P_5 determines that whenever a threshold value of communication cost has been reached, the user should be advised and pre-fetching should be disabled. From then on, objects should only be replicated on-demand by applications.

This set of policies could be installed in the system: i) by default, ii) declaratively defined by an application programmer, iii) setup by a system administrator, or iv) created by the user through an interactive policy generation tool.

This example shows a situation where policies can dynamically manage middleware and application execution, without the need to write new application code for adaptability to each scenario. Furthermore, the applications could have been designed without adaptability in mind altogether. In the example, application code needs simply to be extended in order to allow incremental replication. More sophisticated behavior simply emerges from the concurrent enforcement of this set of policies by Policy-Management in OBIWAN.

IV.2.2 Example Policy

We present now, in greater detail, an example policy that could be used to monitor object replication and network connectivity. This sample policy contains 2 rules, 2 properties, 3 event declarations, and 4 actions, presented next.

```
<?xml version="1.0" encoding="utf-8"?>
<policyxmlns="http://tempuri.org/politica-A1.xsd" name="Test Policy
for Replication Handling" comments="...Testing...">
    <rules>
        <rule>
            <condition eval="==" arg1="property.network.bandwidthClass" arg2="HIGH">
            </condition>
            <eventTriggered name="event.network.bandwidthClassChanged"
                argType="property" argValue="arg1">
            </eventTriggered>
        </rule>
        <rule>
            <condition eval="!=" arg1="property.network.bandwidthClass" arg2="HIGH">
            </condition>
            <eventTriggered name="event.network.bandwidthClassChanged"
                argType="property" argValue="arg1">
            </eventTriggered>
        </rule>
    </rules>
    <properties>
```

```
        <property name="property.system.availableMemory" type="number" value="0"
            assemblyName="PoliPer"
            className="PoliPer.ContextManagement.PropertyUtils._FreeMemory"
            methodName="GetMemoryProperty">
        </property>
        <property name="property.replication.cluster.clusterSize" type="number" value="25">
        </property>
        <property name="property.network.connectivity" type="bool" value="false"
            assemblyName="PoliPer"
            className="PoliPer.ContextManagement.PropertyUtils._NetworkConnectivity"
            methodName="GetNetworkConnectivity">
        </property>
    </properties>
    <events>
        <event name="event.replication.replication-in.object.begin">
        </event>
        <event name="event.replication.replication-in.object.end">
        </event>
        <event name="event.network.bandwidthClassChanged">
        </event>
    </events>
    <actions>
        <action name="anonymous" type="method" assemblyName="PoliPer"
            className="PoliPer.PoliPer_Replication" methodName="ObjectBegin">
            <subscriptions>
                <subscription name="event.replication.replication-in.object.begin" priority="last">
                </subscription>
            </subscriptions>
        </action>
        <action name="anonymous" type="method" assemblyName="PoliPer"
            className="PoliPer.PoliPer_Replication" methodName="ObjectEnd">
            <subscriptions>
                <subscription name="event.replication.replication-in.object.end" priority="last">
                </subscription>
            </subscriptions>
        </action>
        <action name="anonymous" type="method" assemblyName="PoliPer"
            className="PoliPer.PoliPer_Replication" methodName="ObjectBeginEndTrace" multiEvent="true">
            <subscriptions>
                <subscription name="event\.replication\.replication-in\.object\.\w*" priority="last">
                </subscription>
            </subscriptions>
        </action>
        <action name="anonymous" type="method" assemblyName="PoliPer"
            className="PoliPer.PoliPer_Replication" methodName="changeClusterSize">
            <subscriptions>
                <subscription name="event.network.bandwidthClassChanged" priority="last">
                </subscription>
            </subscriptions>
        </action>
    </actions>
</policy>
```

The policy file begins by defining two rules that aim at monitoring variations in network bandwidth. The rules specify two situations, w.r.t. connectivity, that are considered relevant, in this case. They prescribe that an appropriate event (`event.network.bandwidthClassChanged`) be triggered, whenever the available bandwidth (property `property.network.bandwidthClass`) changes to (and from) a specific value (labeled as `"HIGH"`). In the context of this policy, the application and/or the middleware are only interested in being advised when the user device enters, and leaves, a zone with high network bandwidth available. The rules are evaluated when the policy if first loaded and installed in the system, and whenever the entities involved in the conditions are subject to modifications.

Next, the policy installs properties on the context manager that monitor available memory (`property.system.availableMemory`), and connectivity (`property.network.connectivity`), stating their type and initial value. Since these properties are procedural, the policy file also specifies the corresponding associated methods, to be invoked whenever the properties are inspected. As described earlier in Section IV.1.3.1, these methods may be implemented, in portable manner, using GC services and periodic connection attempts, respectively.

The policy files continues by declaring three events it defines. All of them are triggered and handled by code in the OBIWAN library or generated by `obicomp`. They are made public here so that other policies may subscribe to them.

The last part of the policy file is dedicated to specifying actions. OBIWAN code (class `PoliPer.PoliPer_Replication` of the Replication Management module) subscribes to replication events in order to perform actions before and after each object is replicated. Furthermore, it provides an example of an action that subscribes, using regular expressions, to all events regarding object replication to the device. This action can be used for debugging purposes. The last action defines a method to be invoked, when the transitions in available bandwidth, described in the rules, take place. This method will, in turn, modify the current depth of replication clusters. If an action consists of simple attributions of new values to properties, they can also be stated directly, with actions of type `implicit` (defining `methodName` as `internalSET`), together with property name and desired value, as additional attributes.

IV.2.3 Policy-Managed and Adaptive Middleware

Concerning adaptability and support for declarative policies, OBIWAN can be related to several other middleware systems and technologies. This section provides an overview based on their main goals.

Data Representation: There have been proposed several approaches for data representation w.r.t. support for adaptability in applications. Composite Capability and Preferences Profiles (CC/PP), described in (Nilsson et al. 2000; Klyne et al. 2003), define a vocabulary extension of Resource Description Framework (RDF), that uses XML to encode properties and statements regarding web resources (Brickley and Guha 1999; Brickley and Guha 2004). It is used to represent resource capabilities, and presentation preferences, of browsers so that servers are able to adapt web content produced by them, to best suit client devices. A similar approach is also used for expressing agent capabilities in the context of FIPA (www.fipa.org 2002). Tuples are used to represent context information in (Mamei et al. 2003). In (Handorean et al. 2005), they are used for session management and to perform binding to services.

Policies related to web-services can be expressed using the family of WS-Policy schemas (Box et al. 2003; Bajaj et al. 2006), also using XML. This model allows the description of capabilities provided by web-services, as well as constraints imposed by providers and consumers. This allows web-service providers to announce their web-services with extra information that may be queried by consumers performing web-service discovery. Matching information from both facilitates inter-operability.

In OBIWAN, system entities like policies, properties and events are defined in a namespace-based hierarchy combined with regular expressions. A related approach is used for security policy files in the Java language. In Java, permissions are defined in a class hierarchy. Access to system properties can be managed using wildcard substitution when referring to property names. Java policy files are designed solely for security purposes, i.e., granting or denying access to resources. In OBIWAN, the range of policy use is broader. Furthermore, the middleware and applications can react to changes in the system (resource management, access control, etc.) with definable programmatic actions.

Context and Resource Management: The **Context-Toolkit** (Salber et al. 1999; Dey 2001) was one of the first projects to observe the importance of using a uniform approach, when designing context-aware applications (Schilit et al. 1994). Initially, context-aware applications were developed in an ad-hoc manner, by interfacing directly with technologies and hardware used to gather context information (e.g., sensors). The project proposes the analogy between the design of context-aware applications and graphical user interfaces (GUI). Thus, information regarding context is accessed via context-widgets, as user interfaces are via GUI widgets.

Context-widgets are software components that encapsulate context w.r.t. applications, hiding the complexity of interaction with sensors, and converting gathered information to suit the requirements of applications (e.g., returning street names instead of geographical coordinates). Applications can poll a context-widget to obtain its value, or subscribe to be notified of changes in their value. Examples of widgets include *Identity*, *Presence* and *Activity*. Context-widgets improve reusability. They can be combined using Context Aggregators, or mediated using Context Interpreters, in order to provide applications with higher-level context information. Widgets may be distributed and exchange information (context artifacts) using XML. This work does not describe neither how to organize context-widgets (such as a name-space hierarchy), nor how to refer to them using incomplete names (e.g., using regular expressions). It provides a framework to develop context-aware applications but has no support for declarative policies to rule how applications should react to changes in context.

The case for a reflective middleware for context-awareness and adaptability of applications is presented in **CARISMA** (Capra et al. 2003). Context information is comprised of devices, resources, application-defined properties, and user activity and ”mood”. Application developers and users specify the changes in context information considered relevant to them using policies. Middleware adaptability stems from the fact that is configured by meta-data that can be modified (i.e., applications can even change their own policies). The main goal of this work is centered on identifying conflicts in policies (named profiles). Conflicts consist of ambiguities, contradictions, or disagreements found in policies. They may occur internally to a policy, or among policies defined in different devices, applications or by different users.

Conflicts among policies are addressed using an approach based on principles taken from micro-economy. This is based on *utility functions*[1] that are heuristic (defined using an abstract syntax). Utility functions weight parameters such as memory, bandwidth, accuracy, availability, etc. These functions can vary dynamically to reflect changes in user's goals. Based on the utility to each peer, the middleware chooses the policies that maximize the sum of utility values. CARISMA is used to adapt messaging services to variations in the environment (e.g., available bandwidth, battery power) and context information. The middleware automatically chooses message format (with plain, compressed and encrypted text) according to available battery and bandwidth, by evaluating the policies in all participating devices. Message alerts and talk reminders are also adapted (e.g., ring, vibration) according to available battery and whether the user is indoors or outdoors.

Tuples on The Air (Mamei et al. 2003) uses tuples to represent context information and to allow asynchronous communication between applications and components. Tuples are spatially distributed in several tuples-spaces. Tuples are initially injected into the network. They are propagated across nodes, according to application-specific patterns. These patterns determine if tuples are further propagated (forwarded), modified, combined with other tuples, or suppressed. Patterns are implemented as abstract propagation rules parameterized by tuple content. Rules dictate tuple scope, range and direction of propagation. Events are also represented by tuples and published as tuple injections in the distributed tuple-space.

The work described in (Huebscher and McCann 2004; Huebscher and McCann 2006) presents the notion of **Quality of Context (QoC)**, in the context of smart-homes, assuming that the acquisition of context information is a costly operation. It also makes use of utility functions but for a different purpose. When there are multiple sources (e.g., sensors, software components, databases, polling against using recorded values, estimations) where to obtain context information of compatible types, the middleware selects the one most appropriate, i.e., that suits application requirements and minimizes cost (e.g., battery, network communication). Requirements are expressed as Quality of Context constrains, and expressed as function of precision, probability of correctness, resolution/granularity, freshness, and refresh-rate. The middleware chooses the context provider that maximizes satisfaction among applications instead of possibly using multiple providers for the same type of information (each one requested by a different application), thus saving resources.

Gaia (Roman et al. 2002a; Roman et al. 2002b) is a middleware (meta-operating system) that manages context information associated with physical spaces, applications, and users, mapping them on *active spaces*. Each user has its own *virtual space* comprised of *sessions*. Each session manages the association of applications and data with the users. When a user crosses boundaries of active space, Gaia maintains its sessions available, by mapping them to resources available in the new active space.

Gaia allows the use of customization policies, both defined by users as well as default ones provided by the system. All resources in the active space have a XML-description. It provides a high-level scripting language based on Lua (Cerqueira et al. 1999). The very bootstrap of the system is performed by executing a configuration script coded in Lua. Gaia is comprised of an event manager, context service, and a space repository.

[1]In economic science, utility is the value of a good or service, as perceived e.g., by a consumer.

It also includes a context-based file system (Hess 2002), a semantic file system (Gifford et al. 1991) enriched with information gathered from context. Context service is inspired by the Context toolkit (context providers act as context widgets), also allowing higher-level context providers. Context data is expressed as predicate triplets (subject-verb-object). Rules are expressed using first-order logic predicates.

Communication: The work in **Appia** (Miranda et al. 2001; Mocito et al. 2006) aims at adapting communication protocols, used by applications. The variations in context information can be local (e.g., device where the application is running), or distributed (e.g., neighboring devices). Here, adaptability is regarded as the capability to select/switch among communication protocols, according to context, from a pool of protocols previously configured and deployed.

Context information is stored as tuples (context categories). Policies are managed using an economical approach. They are defined by parameters such as rewards, costs, and the probability of occurrence, associated with each transition in context information. Users can not define arbitrary policies with rules. Policies require that all possible states assumed by context information be described upfront, and that all probability matrixes be pre-calculated.

One relevant example of adaptation is the usage of different communication protocols for multi-cast, depending if the device is able to connect to a nearby LAN: i) a best effort protocol that sends a sequence of point-point messages, or ii) a bridging protocol that sends only one point-point message to a gateway node wired to a fixed network that is in charge of relaying messages to other gateways and portable devices. When suitable, protocol ii) saves battery power on the device.

Additional work on adaptability of a communication application is presented in (McFadden et al. 2005). It allows the specification of relevant situations, determined by the evaluation of predicates over selected context properties. When a situation occurs, the appropriate event is triggered. The paper discusses the adaptation of a simple messaging application that is able to, according to context-information, relay messages using different devices and protocols, accordingly to the present location of the message recipient.

The work described in (Handorean et al. 2005) uses context information to perform session management and service binding in ad-hoc networks. Context information is stored using a federated space of Java-tuples. It incorporates the notion of *follow-me session* that masks intervals of disconnection, giving users an appearance of a continuous interaction with a service. In reality, server threads are migrated to servers closer to user's device.

Migration of server threads does not require changes to the virtual machine. It is based on class extension and explicit synchronization check-points provided by the programmer that, when executed, delegate on middleware code that serializes thread data to another server, and restarts the thread. This approach uses similar techniques to those described in (Veiga and Ferreira 2001; Veiga and Ferreira 2002b) (Section II.3.3.9), extending class byte-codes instead of source-code.

In (Curry et al. 2003), a publish/subscribe messaging model is presented, defining channels in a hierarchical manner. This hierarchy is reflective, dynamic and de-coupled from publishers and subscribers. As a consequence, there is no guarantee that a specific channel will exist.

OBIWAN uses a namespace-based organization for entities (including events), that can be accessed hierarchically and, for increased flexibility, with resort to regular expression matching. In OBIWAN, event publishers and subscribers are also fully de-coupled.

Mobile Transactions: With respect to mobile transactions, most research considers networks where mobile hosts connect via wireless to fixed base stations (Barghouti and Kaiser 1991; Ramamritham and Chrysanthis 1996; Walborn and Chrysanthis 1995). These solutions are typically client-server based and do not address the adaptability needs of applications so that they can cope with the multiple requirements and usage diversity found in mobile settings.

However, there are mobile transaction systems that use semantic information to adapt the behavior of transactions. For example, in Pro-Motion (Walborn and Chrysanthis 1995), data is encapsulated in *compacts* allowing the definition of consistency rules to be applied to such data set as a whole. In Clustering (Pitoura and Bhargava 1999), it is possible to specify consistency degrees among replicated data. Moflex (Ku and Kim 2000) also provides a mechanism for describing the associated behavior while crossing wireless cells. With Toggle (Dirckze and Gruenwald 1998), it is possible to specify different atomicity and isolation degrees, by dividing a transaction in vital and non-vital sub-transactions.

Security: One of the most important works addressing policy-management w.r.t. security is Ponder (Dulay et al. 2001). It provides a general-purpose deployment model for security and management policies. Its declarative language is able to express and specify some generic and complex security policies such as role based access control (RBAC) policies. In particular, the obligation policies provided by Ponder are used in an agent platform (Montanari and Tonti 2002) to specify mobility policies of agents. In this platform application logic is completely separated from migration logic.

Negotiation: In (Parlavantzas et al. 2003), resources are encapsulated and managed by an extensible framework. When necessary, resources are dynamically adapted within the middleware to suit each requiring task. This operation is performed through negotiation. The middleware keeps track of the associations among resources and tasks. Resource and context representation in OBIWAN follows a related approach. Nonetheless, in OBIWAN, the focus is somewhat different. Instead of resource management and adaptation, it aims at adapting to resource variations.

QoS non-functional aspects are extracted and defined declaratively by contracts in (Cerqueira et al. 2003). Compatible contracts are combined straightforwardly. When conflicts among requirements from different contracts arise, they are solved based on priority. Connectors effectively externalize interactions and associations between objects. In OBIWAN, a contract can be regarded as a set of policies.

Adaptability and Policy-Management in OBIWAN share some goals and mechanisms with the works just described. Declarative policies are used for flexibility (e.g., negotiation, service binding), and for adaptability to situations known in advance (e.g., different hardware capabilities, service contracts). Context management is used to achieve dynamic adaptability of applications, and the middleware itself, in the presence of variations of the environment, while applications are already running.

IV.2.4 Aggressive Memory Management Techniques

Freeing memory on mobile devices has been addressed previously. In (Messer et al. 2002; Chen et al. 2003b), some objects are migrated to a nearby server machine from where will be re-fetched later, if needed. To provide transparency to applications, such objects are replaced by a surrogate. It imposes changes to the underlying VM. These include i) object tables must account for objects residing in other machines; ii) modifications to CG behavior instrumenting the GC to monitor on an object-by-object basis, which objects to swap-out; and iii) there must be a distributed garbage collection (DGC) algorithm managing references among resident and migrated objects. Even though these approaches are distributed, they do not address replication and related issues (replica management, consistency, etc.).

In (Chen et al. 2003a), a mechanism is proposed to perform compression on the Java virtual machine heap, where large objects (greater than 1.5 Kb) are compressed and decompressed. In addition, large array objects are broken down into smaller sub-objects, each being "lazily allocated" upon its first write access. Constant on-the-fly data compression performed on the heap saves memory but imposes additional CPU load and energy cost, since compression is a computational-intensive process. This solution also imposes the use of a modified VM.

The work in (Chihaia and Gross 2004) describes an analytical model for main memory compression, based exclusively on software. Applying compression to pages in main memory was first suggested in (Wilson 1991). The system reserves a number of pages in main memory that act as an additional intermediate level in memory hierarchy (compressed memory pool). Pages about to be swapped to disk are compressed and swapped to the compressed pool instead. Both page compression and disk writing can be performed asynchronously. Nevertheless, even if compression takes more time than disk writing, the gains obtained during page reload to main memory (decompression is much faster than reading for disk) are much greater.

The only disadvantage is that the compressed-memory pool actually reduces the memory available to applications. Therefore, the system must balance, according to application behavior and needs, the size of the compressed memory pool. Thus, devoting too much memory to the compressed-memory pool hurts performance as much as not reserving enough (Wilson 1991). In multi-core systems, one of the processors may be dedicated solely to compression/decompression activities. This solution is not suited for resource constrained devices as it is directed mainly at workstations. Furthermore it requires modifications to the OS kernel which hinders portability.

The .Net Micro Framework (Thompson and Miller 2006) is a very small .Net virtual machine for embedded systems with very limited resources (e.g., wrist-watches). It employs several techniques to reduce memory foot-print w.r.t. both application code and data. It maintains a global string table that is shared to store names of types, methods and fields, to reduce RAM (w.r.t. application code) and ROM (w.r.t. framework code) usage.

Another innovation provided by .Net Micro is the use of extended weak references. References of this type have precedence over regular weak references. A specialized garbage collector attempts to copy to available persistent memory (e.g., flash-memory in CF[2] and SD[3] cards) unreachable objects that are targeted by extended weak references, instead of reclaiming them, which is the case with objects targeted exclusively by regular weak references. This is performed locally and is therefore limited by the total memory (main, and additional memory cards) of the device.

IV.2.5 Evaluation of Object-Swapping

We presented *Object-Swapping*, a novel approach to reduce memory usage by replicated objects in mobile devices that is portable, and imposes fewer demands on the surrounding infrastructure, w.r.t. other approaches found in previous related work. It is compliant with LGC and DGC managing replicated objects. As a negligible trade-off, our approach requires every reference between objects in different swap-clusters to be mediated by a proxy. Nonetheless, it does not require modification of the underlying virtual machine on the mobile device. It further obviates the need to manage inter-process references among individual resident and swapped-out objects.

This solution has several benefits over a *naive* one that would have one proxy per each object and all references mediated by them. Common application objects are small. So, this could potentially double memory occupation when full-loaded or roughly the same as full. This approach would also inevitably impose a higher performance penalty, due to indirections. Furthermore, even when all objects were swapped, the proxies would still remain. To avoid this, no proxies should exist and the object tables in the VM could be modified to account for swapped objects. This approach still has some overhead but, more importantly, it is not portable.

With object-swapping, devices receiving swapped objects do not need to have VM or middleware installed. The swapping device is instructed simply to store, return or drop XML-data. This favors portability since it does not require swapping devices to install a specific VM, or even a middleware platform like OBIWAN, but simply store portions of text (XML-encoded) data with a corresponding key (that would be the cluster name).

Summary of Chapter and Conclusion: In this chapter, we presented the evaluation of middleware adaptability and object-swapping in OBIWAN. It employed a qualitative approach based on motivating examples and discussion against the related work presented, regarding adaptability and portability.

The evaluation of adaptability in OBIWAN resorted to a number of motivating example situations, which portray variations in the environment that could not be handled by static middleware. These variations can be addressed by OBIWAN, managed by the appropriate policies, which were described. We described a prototypical example policy that monitors available

[2]CompactFlash.
[3]Secure Digital.

memory and network connectivity, in order to adapt the middleware by reconfiguring the parameters of replication mechanisms.

The evaluation of Object-Swapping was focused on portability, w.r.t. avoiding the need to perform modifications on existing virtual machines. Nonetheless, we also addressed the most important aspects that can influence performance, such as memory usage by swapping management itself, and indirections on method invocations. The proposed approach minimizes overhead while being the only one that achieves portability, resorting exclusively to the OBIWAN middleware that executes as user-level code.

In the future, we expect a scenario where (much as wireless access points are becoming omni-present) there will also be an increase in small memory-enabled devices with wireless connectivity, scattered all-over, that are available to any user (either to store data or to relay communications). Our approach is the best suited to this scenario.

V

Conclusion

(this page was intentionally left blank)

V1 Conclusion

I've been waiting for you, Obi-Wan. We meet again at last. The circle is now complete... – in "Star Wars", George Lucas

Part V closes this dissertation. It presents conclusions regarding the whole work presented in the dissertation, and introduces some aspects of on-going and future work.

Presently, programmers are faced with a real hard task when developing mobile distributed applications in which data-sharing is needed. They are forced to deal with system level issues such as data replication, memory management, consistency, durability, availability, security, etc.

The overall goal of this work was to facilitate application development for mobile computing and wide-area networks, i.e., to ease programmers' lives. We proposed an approach based on an object-oriented middleware platform to achieve this. Object-oriented programming is arguably the most widespread paradigm for application development. The main justification for using a middleware approach is obviating the need to modify the underlying operating system and virtual machine. We recall the goals and challenges that motivated this work, enunciated in Chapter I.1, in the form of the following requirements:

1. **Usage of Local Resources:** leverage the usage of existing local resources (CPU, memory) effectively.
2. **Support for Disconnected Work:** minimize dependency on the availability of network connection, e.g., by employing replication techniques.
3. **Transparent Support for Commercial OO Languages**: allow application development using widely adopted object-oriented languages (e.g., C++, Java, C#).
4. **Platform Portability:** impose modifications neither to operating systems nor to dominant commercially available virtual-machines (e.g., Java and .Net).
5. **Enforcement of Referential Integrity:** improve programming reliability by preventing dangling references and memory leaks.
6. **Adaptability**: provide mechanisms to control and adapt resource consumption on running nodes (e.g., memory, network).

The contributions of this dissertation were developed in the context of OBIWAN, a middleware for memory management of replicated objects in distributed and mobile computing. OBIWAN provides a platform that supports the development of applications using object-oriented languages, enabled with object replication, adequate memory management, and adaptability. The OBIWAN middleware platform addresses all the challenges and goals presented in Chapter I.1, summarized in Table V.1.

Project	Usage of Local Resources	Support for Disconnected Work	Transparent Support for Commercial OO Languages	Platform Portability	Enforcement of Referential Integrity	Adaptability
OBIWAN	Yes	Yes	Yes [a]	Yes [b]	Yes[c]	Yes[d]

[a]Java and C#.

[b]OBIWAN prototype implementations execute on unmodified Java, .Net and .Net Compact Framework virtual machines.

[c]Enforced by the algorithms that provide Complete Distributed Garbage Collection: DGC-Consistent Cuts, Algebra-based DCD, and DCC-RM.

[d]OBIWAN replication parameters are ruled by declarative policies that take the variability of the environment into account and adapt to available resources (e.g., available memory and connectivity). W.r.t., memory limitations in mobile constrained devices, *Object-Swapping* in OBIWAN allows transparent swapping-out of useful objects enclosed in swap-clusters, to be reloaded when re-accessed later.

Table V.1.1: Analysis of OBIWAN middleware w.r.t. the goals and challenges presented in Chapter I.1.

The incremental replication mechanisms integrated in the OBIWAN middleware leverage the usage of existing local resources in client machines (1), e.g., CPU and memory, and support disconnected work (2), which were described in Part II.

The OBIWAN middleware allows developers to implement applications by employing commercial object-oriented languages (3), such as C# and Java, to be executed on top of unmodified versions of currently dominant virtual machines (4), such as JVM and .Net CLR.

The OBIWAN middleware enforces referential integrity (5), by employing the DGC algorithms described in Part III. Finally, it enables adaptability of object replication mechanisms and memory management (6), through the use of declarative policies, and object-swapping in mobile constrained devices, which were described in Part IV.

In summary, this dissertation presented contributions in the following areas:

- **Support for Object Replication:** Novel support for object replication in mobile environments without imposing changes to the underlying virtual machine. Incremental object replication and dynamic object clustering provide the necessary flexibility to deal with mobile environments.

- **Distributed Garbage Collection:** Scalable, asynchronous and complete garbage collection for distributed systems with and without replication. Two novel algorithms for distributed garbage collection of distributed object systems, and the first viable solution to achieve complete distributed garbage collection for replicated object systems.

- **Adaptable Object Replication and Memory Management:** A policy-based system applied to management of object replication and memory management, alongside with aggressive memory management mechanisms, such as *Object-Swapping*, to handle memory shortage in mobile devices.

V.1.1 Future Work

The work presented in this dissertation motivates a number of directions for future work. In particular, we plan to address the following subjects:

Performance:

1. A deeper study on the performance, bandwidth usage, latency, and bottlenecks of remote invocation mechanisms (Java RMI, .Net Remoting, and others, both XML and binary-based).

Replication:

2. Implement M-OBIWAN and related prototypes on Java virtual machine running on mobile devices (J2ME, also known as KVM).

Distributed Garbage Collection:

3. investigate how the implementation can be further optimized, namely w.r.t. graph summarization (possibly integrating it with incremental LGC provided with the VM).
4. addressing the formal correctness proof of the algorithms presented.
5. design of a de-centralized version of the DCC-RM algorithm.
6. development of heuristics for determining the minimal set of DeleteScion and DeclareUnreachable messages required to break a distributed cycle in one step.
7. apply the notion of Cycle Detection Algebra to other algorithms, such as those based on back-tracing, obviating the need to maintain state in processes regarding ongoing cycle detections crossing them.
8. apply the DGC algorithms developed to OODBs supporting distribution and replication, such as Ozone and DB4O.

Adaptability:

9. support the concept of a dynamic computing horizon in which resources in a broader sense (memory, disks, printers, internet access, data and even code) can be found in other neighbor devices and used accordingly. The resources available to an application, or mobile agent, should not be restricted to those installed in the running device (computer, PDA, etc.). They should include all that are accessible within acceptable time and hop frames (the *horizon*) from all devices the application/agent is aware of. Proximity-triggered notification should be used to frequently update current horizon definition in each device.
10. develop the concept of a *contextlet*, which encapsulates context information and associated policies (with rules, events, and actions), to be exchanged among neighboring devices.

These directions of possible future work range from non-core performance and implementation aspects, usually addressable in the context of Final Year Projects and M.Eng. degrees (e.g., 1, 2), to more theoretical work to be addressed either by the author (e.g., 4, 6), or in the context of future M.Sc. (e.g., 3, 8, 9, 10) and Ph.D. (e.g., 5, 7) dissertations.

A Biblographic References

Abdullahi, S. E. and G. A. Ringwood (1998). Garbage collecting the internet: a survey of distributed garbage collection. *ACM Computing Surveys (CSUR) 30*(3), 330–373.

Abiteboul, S., A. Bonifati, G. Cobena, I. Manolescu, and T. Milo (2003). Dynamic xml documents with distribution and replication. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, New York, NY, USA, pp. 527–538. ACM Press.

Ahuja, S., N. Carriero, and D. Gelernter (1986). Linda and friends. *Computer 19*(8), 26–34.

Alliance, T. O. (2003). *OSGi Service Platform, Release 3*. IOS Press.

Amsaleg, L., P. Ferreira, M. Franklin, and M. Shapiro (1995a, October). Evaluating garbage collectors for large persistent stores. In *OOPSLA'95 W'shop on Object Database Behavior, Benchmarks, and Performance*, Austin, TX.

Amsaleg, L., M. Franklin, and O. Gruber (1995b, September). Efficient incremental garbage collection for client–server object database systems. In *Twenty-first Int'l Conf. on Very Large Databases (VLDB95)*, Zurich, Switzerland.

Amsaleg, L., M. J. Franklin, and O. Gruber (1999). Garbage collection for a client-server persistent object store. *ACM Trans. Comput. Syst. 17*(3), 153–201.

Andrade, N., L. Costa, G. Germoglio, and W. Cirne (2005). Peer-to-peer grid computing with the OurGrid Community. *Proceedings of the 23rd Brazilian Symposium on Computer Networks*.

Andrews, K., F. Kappe, and H. Maurer (1995, april). The hyper-g network information systems. *J.UCS 1(4)*.

Androutsellis-Theotokis, S. and D. Spinellis (2004). A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv. 36*(4), 335–371.

Apache Foundation (2002). Apache object relational bridge - OJB.

Apache Software Foundation (1997, july). Module mod_rewrite, url rewriting engine.

Appel, A. W., J. R. Ellis, and K. Li (1988, June). Real-time concurrent collection on stock multiprocessors. In *Proc. of the SIGPLAN'88 Conf. on Programming Language Design and Implementation*, Atlanta GA (USA), pp. 11–20. ACM.

Archer, A. W. T. (2002). *Inside C#* (2 ed.). Microsoft Press.

Arnold, K. and J. Gosling (1996). *The Java Programming Language*. Addison-Wesley.

Atkinson, M. P., F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik (1989). The object-oriented database system manifesto. In *DOOD*, pp. 223–240.

Azatchi, H. and E. Petrank (2003). Integrating generations with advanced reference counting garbage collectors. In *CC*, pp. 185–199.

Bacon, D., S. Fink, and D. Grove (2002). Space-and Time-Efficient Implementation of the Java Object Model. *Proceedings of the 16th European Conference on Object-Oriented Programming*, 111–132.

Bacon, D. F., C. R. Attanasio, H. Lee, V. T. Rajan, and S. Smith (2001). Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 92–103.

Bacon, D. F., P. Cheng, and V. T. Rajan (2004). A unified theory of garbage collection. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, pp. 50–68. ACM Press.

Bacon, J., J. Bates, R. Hayton, and K. Moody (1995). Using Events to Build Distributed Applications. *2nd International Workshop on Services in Distributed and Networked Environments.*

Bacon, J., K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, and M. Spiteri (2000). Generic support for distributed applications. *Computer 33*(3), 68–76.

Baden, S. B. (1983). Low-overhead storage reclamation in the Smalltalk-80 virtual machine. In G. Krasner (Ed.), *Smalltalk-80: Bits of History, Words of Advice*, pp. 331–342. Addison-Wesley.

Baduel, L., F. Baude, and D. Caromel (2002). Efficient, flexible, and type group communication in java. In *Proc. of ACM Joint ACM Java Grande - ISCOPE 2002 Conference (JGI'02).*

Bajaj, S., D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam-Baker, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, H. Prafullchandra, C. von Riegen, D. Roth, J. Schlimmer, C. Sharp, J. Shewchuk, A. Vedamuthu, Ümit Yalçýnalp, and D. Orchard (2006). Web Services Policy Framework (WS-Policy) Version 1.2. *BEA Systems Inc., International Business Machines Corporation, Microsoft Corporation, Inc., SAP AG, Sonic Software, and VeriSign Inc..*

Baker, H. G. (1978). List processing in real-time on a serial computer. *Comm. of the ACM 21*(4), 280–94. Also AI Laboratory Working Paper 139, 1977.

Baker, M., R. Buyya, and D. Laforenza (2002). Grids and Grid technologies for wide-area distributed computing. *Software-Practice and Experience 32*(15), 1437–66.

Bal, H. E., M. F. Kaashoek, and A. S. Tanenbaum (1992). Orca: A language for parallel programming of dist. systems. *ACM Trans. on Software Engineering 18*(3), 190–205.

Bal, H. E. and A. S. Tanenbaum (1990, may). Orca: A language for distributed object-based programming. *SIGPLAN Notices 25*(5), 17–24.

Baratloo, A., P. E. Chung, Y. Huang, S. Rangarajan, and S. Yajnik (1998). Filterfresh: Hot replication of java rmi server objects. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. USENIX.

Barbara, D. (1999). Mobile computing and databases - a survey. *IEEE Transactions on Knowledge and Data Engineering 11*(1), 108–117.

Barghouti, N. S. and G. E. Kaiser (1991). Concurrency control in advanced database applications. *ACM Computing Surveys 23*(3), 269–317.

Bartlett, J. F. (1988, February). Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Western Research Laboratory, Palo Alto, CA (USA).

Bekkers, Y. and J. Cohen (Eds.) (1992, 16–18 September). *Proc. of Int'l W'shop on Memory Management*, Volume 637 of *Lecture Notes in Computer Science*, St Malo, France. Springer-Verlag.

Bergman, M. K. (2001). The deep web: Surfacing hidden value. *The Journal of Electronic Publishing 7*(1).

Berners-Lee, T. (1994, June). Universal resource identifiers in www - request for comments: 1630.

Berners-Lee, T., R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret (1994). The world-wide web. *Commun. ACM 37*(8), 76–82.

Bershad, B. N. and M. J. Zekauskas (1991, September). Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, Pittsburgh, PA (USA).

Bevan, D. I. (1987, June). Distributed garbage collection using reference counting. In *Parallel Arch. and Lang. Europe*, Eindhoven, The Netherlands, pp. 117–187. Spring-Verlag Lecture Notes in Computer Science 259.

Bhushan, A. (1971). File transfer protocol request for comments: 114.

Birrell, A., D. Evers, G. Nelson, S. Owicki, and E. Wobber (1993a, December). Dist. garbage collection for network objects. Technical Report 116, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301.

Birrell, A. and B. J. Nelson (1984). Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*(1), 39–59.

Birrell, A., G. Nelson, S. Owicki, and E. Wobber (1993b). Network objects. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, New York, NY, USA, pp. 217–230. ACM Press.

Bishop, P. B. (1977, May). Computer systems with a very large address space and garbage collection. MIT Report LCS/TR–178, Laboratory for Computer Science, MIT, Cambridge, MA.

Blackburn, S. and K. S. McKinley (2003). Ulterior reference counting: fast garbage collection without a long wait. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pp. 344–358. ACM.

Blackburn, S. M., P. Cheng, and K. S. McKinley (2004a). Myths and realities: the performance impact of garbage collection. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, pp. 25–36. ACM.

Blackburn, S. M., P. Cheng, and K. S. McKinley (2004b). Oil and water? high performance garbage collection in java with mmtk. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, pp. 137–146. IEEE Computer Society.

Blondel, X., P. Ferreira, and M. Shapiro (1998). Implementing garbage collection in the perdis system. In *Proc. of the Eigth Int'l W'shop on Persistent Object Systems: Design, Implementation and Use (POS-8)*.

Boehm, H.-J. (1991, October). Simple GC-safe compilation. In P. R. Wilson and B. Hayes (Eds.), *OOPSLA/ECOOP '91 W'shop on Garbage Collection in Object-Oriented Systems*.

Boehm, H.-J. (1993, June). Space efficient conservative garbage collection. In *Proc. of SIGPLAN'93 Conf. on Programming Languages Design and Implementation*, Volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque, New Mexico, pp. 197–206. ACM Press.

Boehm, H.-J., A. J. Demers, and S. Shenker (1991, June). Mostly parallel garbage collection. In *Proc. of the SIGPLAN'91 Conf. on Programming Language Design and Implementation*, Toronto (Canada), pp. 157–164. ACM.

Boehm, H.-J. and M. Weiser (1988, September). Garbage collection in an uncooperative environment. *Software: Practice and Experience 18*(9), 807–820.

Box, D., F. Curbera, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, and J. Shewchuk (2003). Web Services Policy Framework (WS-Policy). *joint specification by BEA Systems, IBM, and Microsoft, May*.

Box, D., D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer (2000). Simple object access protocol (soap) 1.1, w3c note 08. Technical report, World Wide Web Consortium, 2000.

Box, D. and A. Hejlsberg (2006, may). The linq project: .net language integrated query.

Braeutigam, F., G. Mueller, P. Nyfelt, and L. Mekenkamp (2002). Ozone - java oodbms. www.ozone-db.org.

Brickley, D. and R. Guha (1999). Resource description framework (rdf) schema specification. Technical report, W3C Proposed Recommendation 03 March.

Brickley, D. and R. Guha (2004). Rdf vocabulary description language 1.0: Rdf schema. Technical report, W3C Recommendation 10 February.

Brodie-Tyrrell, W., H. Detmold, K. E. Falkner, and D. S. Munro (2004). Garbage collection for storage-oriented clusters. In V. Estivill-Castro (Ed.), *ACSC*, Volume 26 of *CRPIT*, pp. 99–108. Australian Computer Society.

Brooch, G. (1993). *Object-Oriented Analysis and Design with Applications* (2nd ed.). Addison-Wesley Professional.

Brooks, R. A. (1984, August). Trading data space for reduced time and code space in real-time garbage collection on stock hardware. See Steele (1984), pp. 256–262.

Butrico, M., H. Chang, A. Cocchi, N. Cohen, D. Shea, and S. Smith (1997). Gold rush: Mobile transaction middleware with java-object replication. In *3rd Usenix Conference on Object-Oriented, Technologies*. Usenix.

Butterwoth, P., A. Otis, and J. Stein (1991, October). The GemStone object database management system. *Communications of the ACM 34*(10), 64–77.

Cabri, G., L. Leonardi, and F. Zambonelli (2000). MARS: a programmable coordination architecture for mobile agents. *IEEE Internet Computing 4*(4), 26–35.

Capra, L., W. Emmerich, and C. Mascolo (2001). Middleware for mobile computing: Awareness vs. transparency. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pp. 164. IEEE Computer Society.

Capra, L., W. Emmerich, and C. Mascolo (2003). CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering 29*(10), 929–944.

Carey, M. J. and D. DeWitt (1986, September). The architecture of the EXODUS extensible DBMS. In *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA (USA), pp. 52–65. IEEE.

Carey, M. J., D. J. DeWitt, and J. F. Naughton (1993). The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data) 22*(2), 12–21.

Carlsson, S., C. Mattsson, and M. Bengtsson (1990, October). A fast expected-time compacting garbage collection algorithm. See Jul and Juul (1990).

Carriero, N. and D. Gelernter (1986). The s/net's linda kernel. *ACM Trans. Comput. Syst. 4*(2), 110–129.

Carzaniga, A., D. Rosenblum, and A. Wolf (2000). Achieving scalability and expressiveness in an Internet-scale event notification service. *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, 219–227.

Castro, M., A. Adya, B. Liskov, and A. C. Meyers (1997). Hac: hybrid adaptive caching for distributed storage systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, New York, NY, USA, pp. 102–115. ACM Press.

Castro, M., P. Druschel, A. Kermarrec, and A. Rowstron (2002). Scribe: a large-scale and decentralised application-level multicast infrastructure, IEEE J. *Selected Areas Commun.(JSAC)(Special issue NetworkSupport Multicast Commun.) 20*(8), 100–110.

Caughey, S. J., D. Hagimont, and D. B. Ingham (2000, February). Deploying distributed objects on the Internet. *Recent Advances in Dist. Systems, Springer Verlag LNCS, Eds. S. Krakowiak and S.K. Shrivastava 1752*.

Cederqvist, P. et al. (2002). Version Management with CVS.

Cerqueira, R., S. Ansaloni, O. Loques, and A. Sztajnberg (2003, june). Deploying non-functional aspects by contract. In *The 2nd International Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil.

Cerqueira, R., C. Cassino, and R. Ierusalimschy (1999). Dynamic component gluing across different componentware systems. In *DOA*, pp. 362–371.

Chen, D., A. Messer, D. Milojicic, and S. Dwarkadas (2003b). Garbage collector assisted memory offloading for memory-constrained devices. In *Fifth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Press.

Chen, G., M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko (2003a). Heap compression for memory-constrained java environments. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 282–301. ACM Press.

Cheney, C. J. (1970, November). A non-recursive list compacting algorithm. *Comm. of the ACM 13*(11), 677–8.

Chiang, C.-Y., M. T. Liu, and M. E. Muller (1999, September). Caching neighborhood protocol: a foundation for building dynamic web caching hierarchies with proxy servers. In *International Conference on Parallel Processing*.

Chihaia, I. and T. Gross (2004). An analytical model for software-only main memory compression. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, New York, NY, USA, pp. 107–113. ACM Press.

Chockler, G., D. Dolev, R. Friedman, and R. Vitenberg (2000, April). Implementing caching service for dist. corba objects. In *Proc. of the IFIP/ACM Int. Conf. on Dist. Systems Platforms and Open Dist. Processing (Middleware'2000) - Springer Verlag*, Heidelberg.

Christopher, T. W. (1984, June). Reference count garbage collection. *Software Practice and Experience 14*(6), 503–507.

Clark, D. W. (1976, June). An efficient list moving algorithm using constant workspace. *Comm. of the ACM 19*(6), 352–354.

Cohen, J. (1981, September). Garbage collection of linked data structures. *Computing Surveys 13(3)*, 341–367.

Cohen, J. and L. Trilling (1967). Remarks on garbage collection using a two level storage. *BIT 7*(1), 22–30.

Collins, G. E. (1960, December). A method for overlapping and erasure of lists. *Comm. of the ACM 3*(12), 655–657.

Collins, G. E. (1961, October). Experience in automatic storage allocation. *Comm. of the ACM 4*(10), 436–440.

Cook, J. E., A. L. Wolf, and B. G. Zorn (1994, May). Partition selection policies in object database garbage collection. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, Minneapolis MN (USA), pp. 371–382. ACM SIGMOD.

Cook, J. E., A. L. Wolf, and B. G. Zorn (1998). A highly effective partition selection policy for object database garbage collection. *IEEE Trans. Knowl. Data Eng. 10*(1), 153–172.

Courtney, J. (2001, mar). J2me connected configuration (cdc) version 1.0 (jsr 36): Final release. Technical report, Sun Microsystems.

Courtney, J. (2005, aug). J2me connected configuration (cdc) version 1.1 (jsr 218): Final release. Technical report, Sun Microsystems.

Creech, M. L. (1996, May). Author-oriented link management. In *Fifth International WWW Conference*, France.

Crow, B., I. Widjaja, L. Kim, and P. Sakai (1997). IEEE 802.11 Wireless Local Area Networks. *Communications Magazine, IEEE 35*(9), 116–126.

Cugola, G., E. Di Nitto, and A. Fuggetta (2001). The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering 27*(9), 827–850.

Cunningham, W. and B. Leuf (2001). *The Wiki Way. Quick Collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Curran, K., K. Doherty, and R. Power (2004). WikiWikiWeb as a Tool for Collaboration. *Information Technology Journal 3*(2), 206–210.

Curry, E., D. Chambers, and G. Lyons (2003, june). Introducing reflective techniques to message hierarchies. In *The 2nd International Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil.

Date, C. J. (1999). *An Introduction to Database Systems* (7 ed.). Addison Wesley.

Davies, N., A. Friday, S. P. Wade, and G. S. Blair (1998a). An asynchronous distributed systems platform for heterogeneous environments. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, New York, NY, USA, pp. 66–73. ACM Press.

Davies, N., A. Friday, S. P. Wade, and G. S. Blair (1998b). L2imbo: A distributed systems platform for mobile computing. *MONET 3*(2), 143–156.

Davies, N., S. Wade, A. Friday, and G. Blair (1997, May). Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications. In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, Toronto, Canada, pp. 291–302.

db4objects, Inc. db4o :: Native java & .net object database :: Open source.

Demers, A., M. Weiser, B. Hayes, D. G. Bobrow, and S. Shenker (1990, January). Combining generational and conservative garbage collection: Framework and implementations. In *Conf. Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, San Francisco, CA, pp. 261–269. ACM Press.

Demers, A. J., K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch (1994, 8-9). The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, Santa Cruz, California, pp. 2–7.

Detlefs, D. (1992, August). Garbage collection and run-time typing as a C++ library. In *C++ Conference*, Portland OR (USA), pp. 37–56. Usenix.

Detlefs, D. L. (1990, October). Concurrent, atomic garbage collection. See Jul and Juul (1990).

Detlefs, D. L. (1991, November). *Concurrent, Atomic Garbage Collection*. Ph. D. thesis, Dept. Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213.

Detlefs, D. L., P. A. Martin, M. Moir, and J. Guy L. Steele (2001). Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing, August 2001.*

Detlefs, D. L., P. A. Martin, M. Moir, and J. Guy L. Steele (2002). Lock-free reference counting. *Distrib. Comput. 15*(4), 255–271.

Deutsch, L. P. and D. G. Bobrow (1976, September). An efficient incremental automatic garbage collector. *Comm. of the ACM 19*(9), 522–526.

Deux, O. et al. (1990). The Story of O2. *IEEE Transactions on Knowledge and Data Engineering 2*(1), 91–108.

Deux, O. et al. (1991, October). The O_2 system. *Communications of the ACM 34*(10), 34–48.

Dey, A. (2001). Understanding and Using Context. *Personal and Ubiquitous Computing 5*(1), 4–7.

Dickman, P. (1991, September). *Distributed Object Management in a Non-Small Graph of Autonomous Networks With Few Failures*. Ph. D. thesis, University of Cambridge, United Kingdom.

Dieckmann, S. and U. Holzle (1999). A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. *Proceedings of the 13th European Conference on Object-Oriented Programming*, 92–115.

Dijkstra, E. W., L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens (1978, November). On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM 21*(11), 966–975.

Dirckze, R. A. and L. Gruenwald (1998). A toggle transaction management technique for mobile multidatabases. In *Proceedings of the CIKM 98*, Bethesda, MD, USA, pp. 371–377.

Domani, T., E. K. Kolodner, E. Lewis, E. E. Salant, K. Barabash, I. Lahan, Y. Levanoni, E. Petrank, and I. Yanover (2000). Implementing an on-the-fly garbage collector for java. In *ISMM*, pp. 155–166.

Dulay, N., E. Lupu, M. Sloman, and N. Damianou (2001). A policy deployment model for the ponder language. In *7th IEEE/IFIP International Symposium on Integrated Network Management*, Seattle, USA. IEEE press.

Eckstein, R. (2001). *XML Pocket Reference* (2 ed.). OReilly.

Edelson, D. R. (1992a, January). A mark-and-sweep collector for C++. In *Proc. 19th Symp. on Principles of Programming Lang.*, Albuquerque, NM (USA), pp. 51–57. ACM SIGPLAN-SIGACT.

Edelson, D. R. (1992b, August). Smart pointers: They're smart, but they're not pointers. In *C++ Conference*, Portland, OR (USA), pp. 1–19. Usenix.

EJB 3.0 Expert Group (2006, may). Enterprise javabeans(tm),version 3.0 (jsr 220). Technical report, Sun Microsystems.

Englander, R. and M. Loukides (1997). *Developing Java Beans*. O'Reilly & Associates. ISBN: 1565922891.

Eric Freeman, Susanne Hupfer, K. A. (1999). *JavaSpaces(TM) Principles, Patterns, and Practice*. Pearson Education.

Esposito, D. (2004, feb). A first look at objectspaces in visual studio 2005, discover the capabilities of objectspaces, the .net object/relational mapping framework. Technical report, Microsoft.

Çetintemel, U., P. J. Keleher, B. Bhattacharjee, and M. J. Franklin (2003). Deno: A decentralized, peer-to-peer object-replication system for weakly connected environments. *IEEE Trans. Comput. 52*(7), 943–959.

Eugster, P. T., P. A. Felber, R. Guerraoui, and A.-M. Kermarrec (2003). The many faces of publish/subscribe. *ACM Comput. Surv. 35*(2), 114–131.

Fenichel, R. R. and J. C. Yochelson (1969, November). A Lisp garbage collector for virtual memory computer systems. *Comm. of the ACM 12*(11), 611–612.

Ferreira, P. (1991, October). Reclaiming storage in an object oriented platform supporting extended C++ and Objective-C applications. In *Proc. of the Int'l W'shop on Object Orientation in Operating Systems*, Palo Alto (USA).

Ferreira, P. and M. Shapiro (1993, December). Distribution and persistence in multiple and heterogeneous address spaces. In *Proc. Int. Workshop on Object-Orientation in Operating Systems*, Asheville NC (USA).

Ferreira, P. and M. Shapiro (1994a, November). Garbage collection and DSM consistency. In *First Symposium on Operating Systems Design and Implementation*, Monterey, CA, pp. 229–241. ACM Press.

Ferreira, P. and M. Shapiro (1994b, September). Garbage collection of persistent objects in dist. shared memory. In *Proc. of the 6th Int'l W'shop on Persistent Object Systems*, Tarascon (France). Springer-Verlag.

Ferreira, P. and M. Shapiro (1995, August). Garbage collection in the larchant persistent dist. shared store. In *Proc. of the 5th W'shop on Future Trends in Dist. Computing Systems*, Cheju Island (Republic of Koreia). IEEE.

Ferreira, P. and M. Shapiro (1996, May). Larchant: Persistence by reachability in dist. shared memory through garbage collection. In *Sixteenth Int'l Conf. on Dist. Computer Systems*, Hong Kong.

Ferreira, P. and M. Shapiro (1998, July). Modelling a distributed cached store for garbage collection: the algorithm and its correctness proof. In *ECOOP'98, Proc. of the Eight European Conf. on Object-Oriented Programming*, Brussels (Belgium).

Ferreira, P., M. Shapiro, X. Blondel, O. Fambon, J. ao Garcia, S. Kloosterman, N. Richer, M. Robert, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak (2000, February). PerDiS: design, implementation, and use of a PERsistent DIstributed Store. *Recent Advances in Dist. Systems, Springer Verlag LNCS, Eds. S. Krakowiak and S.K. Shrivastava 1752*.

Ferreira, P. and L. Veiga (2005, july). Garbage collection curriculum. Msdn academic alliance curriculum repository, object id 6812, Microsoft.

Ferreira, P., L. Veiga, and C. Ribeiro (2003, November). Obiwan - design and implementation of a middleware platform. *IEEE Transactions on Parallel and Distributed Systems 14(11)*, 1086–1099.

Fessant, F. L. (2001, August). Detecting distributed cycles of garbage in large-scale systems. In *Conference on Principles of Distributed Computing(PODC)*.

Fidler, E., H.-A. Jacobsen, G. Li, and S. Mankovski (2005). The padres distributed publish/subscribe system. In S. Reiff-Marganiec and M. Ryan (Eds.), *FIW*, pp. 12–30. IOS Press.

Fischer, M. J. and A. Michael (1982). Sacrificing serializability to attain high availability of data in an unreliable network. In *PODS '82: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, New York, NY, USA, pp. 70–75. ACM Press.

Fisher, M., N. Lynch, and M. Patterson (1985, April). Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*(2), 274–382.

Fok, C., G. Roman, and G. Hackmann (2004). A lightweight coordination middleware for mobile computing. *Proceedings of the 6th Internation Conference on Coordination Models and Languages (Coordination 2004)*, 135–151.

Foster, I. (1989). A multicomputer garbage collector for a single-assignment language. *Int'l Journal of Parallel Programming 18*(3), 181–203.

Foster, I. and C. Kesselman (1997). Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications 11*(2), 115–128.

Foster, I. and C. Kesselman (1998). *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

Frenot, S., M. Avin, and N. Almasri (2002). EJB components Migration Service and Automatic Deployment. *INRIA Rapport de Recherche 4480*.

Fuchs, M. (1995, September). Garbage collection on an open network. In H. Baker (Ed.), *Proc. of Int'l W'shop on Memory Management*, Volume 986 of *Lecture Notes in Computer Science*, Concurrent Engineering Research Center, West Virginia University, Morgantown, WV. Springer-Verlag.

Gelernter, D. (1985). Generative communication in linda. *ACM Trans. Program. Lang. Syst. 7*(1), 80–112.

General Magic, I. (1997). Introduction to the odyssey api. http://www.genmagic.com/agents/odyssey.html.

Gifford, D. K., P. Jouvelot, M. A. Sheldon, and J. W. O. Jr (1991). Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 16–25. Association for Computing Machinery SIGOPS.

Glaser, H. W. (1987). On minimal overhead reference count garbage collection in dist. systems. Technical report, Department of Computing, Imperial College, London.

Glaser, H. W. and P. Thompson (1987, January). Lazy garbage collection. *Software Practice and Experience 17*(1), 1–4.

Goldberg, B. (1989, June). Generational reference counting: A reduced-communication distributed storage reclamation scheme. In *Programming Languages Design and Implementation*, Number 24(7) in SIGPLAN Notices, Portland OR (USA), pp. 313–321. SIGPLAN: ACM Press.

Gray, J. N. and A. Reuter (1993). *Transaction Processing: Concepts*. Morgan Kaufmann.

Gray, R. S. (1995, December). Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland.

Greif, I. (Ed.) (1988). *Computer-Supported Cooperative Work: A Book of Readings*. MORGAN KAUFFMAN.

Grimm, R., T. Anderson, B. Bershad, and D. Wetherall (2000). A system architecture for pervasive computing. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pp. 177–182. ACM Press.

Grimm, R., J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall (2004). System support for pervasive applications. *ACM Trans. Comput. Syst. 22*(4), 421–486.

Gruber, R., F. Kaashoek, B. Liskov, and L. Shrira (1994, December). Disconnected operation in thor object-oriented database system. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA.

Gupta, A. and W. K. Fuchs (1993). Garbage collection in a distributed object-oriented system. *IEEE Transactions on Knowledge and Data Engineering 5*(2), 257–265.

Guy, R. G., P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek (1999). Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining*, pp. 254–265. Springer-Verlag.

Haahr, M., R. Cunningham, and V. Cahill (2000, December). Towards a generic architecture for mobile object-oriented applications.

Haartsen, J., E. BV, and N. Emmen (2000). The Bluetooth radio system. *Personal Communications, IEEE [see also IEEE Wireless Communications] 7*(1), 28–36.

Hagimont, D. and F. Boyer (2001, January). A configurable RMI mechanism for sharing distributed Java objects. *IEEE Internet Computing 5*.

Handorean, R., R. Sen, G. Hackmann, and G.-C. Roman (2005). Context aware session management for services in ad hoc networks. In *IEEE International Conference on Services Computing*, pp. 113–120.

Hayes, B. (1991, October). Using key object opportunism to collect old objects. In A. Paepcke (Ed.), *OOPSLA'91 ACM Conf. on Object-Oriented Systems, Languages and Applications*, Volume 26(11) of *ACM SIGPLAN Notices*, Phoenix, Arizona, pp. 33–46. ACM Press.

Hayton, R., J. Bacon, J. Bates, and K. Moody (1996). Using events to build large scale distributed applications. *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, 9–16.

Herlihy, M. and J. E. B. Moss (1992, May). Lock-free garbage collection for multiprocessors. *IEEE Trans. on Parallel and Dist. Systems 3*(3).

Hertel, C. and C. R. Hertel (2003, aug). *Implementing CIFS: The Common Internet File System* (1 ed.). Prentice Hall PTR.

Hess, C. (2002). A Context File System for Ubiquitous Computing Environments. *University of Illinois at Urbana-Champaign, Urbana-Champaign, CS Technical Report UIUCDCS-R-2002-2285 UILU-ENG-2002-1729, July 2002.*

Hosking, A. L. and J. E. B. Moss (1993, September). object fault handling for persistent programming languages: a perfromance evaluation. In *ACM Conf. on Object-Oriented PRogramming Systems, Languages and Applications, 288-303.*

HostPulse (2002). Broken-link checker, www.hostpulse.com.

Howard, J. H., M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West (1988). Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*(1), 51–81.

hua Chu, H., H. Song, C. Wong, S. Kurakake, and M. Katagiri (2004). Roam, a seamless application framework. *J. Syst. Softw. 69*(3), 209–226.

Huang, X., S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng (2004). The garbage collection advantage: improving program locality. In J. M. Vlissides and D. C. Schmidt (Eds.), *OOPSLA*, pp. 69–80. ACM.

Huang, Y. and H. Garcia-Molina (2004). Publish/subscribe in a mobile environment. *Wirel. Netw. 10*(6), 643–652.

Hudak, P. R. and R. M. Keller (1982, August). Garbage collection and task deletion in dist. applicative processing systems. In *Conf. Record of the 1982 ACM Symposium on Lisp and Functional Programming*, Pittsburgh, PA, pp. 168–178. ACM Press.

Hudson, R., R. Morrison, J. E. B. Moss, and D. Munro (1997, October). Garbage collecting the world: One car at time. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta (U.S.A.).

Hudson, R. L. and J. E. B. Moss (1992, 16–18 September). Incremental garbage collection for mature objects. See Bekkers and Cohen (1992).

Huebscher, M. and J. McCann (2004). Adaptive middleware for context-aware applications in smart-homes. *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, 111–116.

Huebscher, M. C. and J. McCann (2006). An adaptive middleware framework for context-aware applications. *Personal and Ubiquitous Computing 10*(1), 12–20.

Huelsbergen, L. and J. R. Larus (1993, May). A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, Volume 28(7) of *ACM SIGPLAN Notices*, San Diego, CA, pp. 73–82. ACM Press.

Hughes, J. (1985, September). A distributed garbage collection algorithm. In J.-P. Jouannaud (Ed.), *Functional Languages and Computer Architectures*, Number 201 in Lecture Notes in Computer Science, Nancy (France), pp. 256–272. Springer-Verlag.

IBM (2003). Enterprise tspaces. (available at http://www.almaden.ibm.com/cs/TSpaces/ets.html).

Ichisuki, Y. and A. Yonezawa (1990, October). Dist. garbage collection using group reference counting. See Jul and Juul (1990).

Ingham, D., S. Caughey, and M. Little (1996). Fixing the "Broken-Link" problem: the W3Objects approach. *Computer Networks and ISDN Systems 28*(7–11), 1255–1268.

Ingham, D. B., M. C. Little, S. J. Caughey, and S. K. Shrivastava (1995). W3Objects: Bringing object-oriented technology to the Web. *World-Wide Web Journal 1*.

ISO (1986). Iso 8879:1986 - information processing – text and office systems – standard generalized markup language (sgml). Technical report.

Iverson, W. (2004). Hibernate: A J2EE (TM) Developer's Guide.

Iyer, S., A. Rowstron, and P. Druschel (2002). Squirrel: A decentralized peer-to-peer web cache.

Java Ranch (2001). Java ranch. http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=26&t=000522.

Jing, J., A. S. Helal, and A. Elmagarmid (1999). Client-server computing in mobile environments. *ACM Comput. Surv. 31*(2), 117–157.

Jones, R. (1999). *the* garbage collection bibliography. http://www.cs.kent.ac.uk/people/staff/rej/gcbib/gcbib.html.

Jones, R. and R. Lins (1996). *Garbage Collection, Algorithms for Automatic Dynamic Memory Management*. Chichester (GB): Wiley. ISBN 0-471-94148-4.

Jones, R. E. and R. D. Lins (1992, December). Cyclic weighted reference counting without delay. Technical Report 28–92, Computing Laboratory, The University of Kent at Canterbury.

Joseph, A. D., A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek (1995, December). Rover: A toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Co., pp. 156–171.

Joseph, A. D., J. A. Tauber, and M. F. Kaashoek (1997). Mobile computing with the rover toolkit. *IEEE Transactions on Computers 46*(3), 337–352.

Jul, E. and N.-C. Juul (Eds.) (1990, October). *OOPSLA/ECOOP '90 W'shop on Garbage Collection in Object-Oriented Systems*, Ottawa.

Juul, N. C. and E. Jul (1992, September). Comprehensive and robust garbage collection in a distributed system. In *Proc. Int. Workshop on Memory Management*, Number 637 in Lecture Notes in Computer Science, Saint-Malo (France), pp. 103–115. Springer-Verlag.

Kafura, D. G., M. Mukherji, and D. Washabaugh (1995). Concurrent and distributed garbage collection of active objects. *IEEE Trans. Parallel Distrib. Syst. 6*(4), 337–350.

Kafura, D. G., D. Washabaugh, and J. Nelson (1990). Garbage collection of actors. In *OOPSLA/ECOOP*, pp. 126–134.

Kapitza, R., H. Schmidt, and F. J. Hauck (2005). Platform-independent object migration in corba. In *OTM Conferences (1)*, pp. 900–917.

Kappe, F. (1995, February). A Scalable Architecture for Maintaining Referential Integrity in Distributed Information Systems. *Journal of Universal Computer Science 1*(2).

Keleher, P., A. Cox, and W. Zwaenepoel (1994, January). TreadMarks: Dist. shared memory on standard workstations and operating systems. *Proc. of the 1994 Winter USENIX Conf.*.

Keleher, P., A. L. Cox, and W. Zwaenepoel (1992, May). Lazy release consistency for software distributed shared memory. In *Proc. 19th Int. Symposium on Comp. Architecture*, Gold Coast (Australia), pp. 13–21.

Keleher, P. J. and U. Çetintemel (2000). Consistency management in deno. *Mobile Networks and Applications 5*(4), 299–309.

Kermarrec, A., A. Rowstron, M. Shapiro, and P. Druschel (2001, August). The icecube approach to the reconciliation of divergent replicas. In *20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, Newport, RI, USA.

Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997). Aspect-oriented programming. In M. Akşit and S. Matsuoka (Eds.), *Proceedings European Conference on Object-Oriented Programming*, Volume 1241, pp. 220–242. Berlin, Heidelberg, and New York: Springer-Verlag.

Kim, J. and Y. Hsu (2000). Memory system behavior of Java programs: methodology and analysis. *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 264–274.

Kistijantoro, A. I., G. Morgan, S. K. Shrivastava, and M. C. Little (2003). Component replication in distributed systems: A case study using enterprise java beans. In *SRDS*, pp. 89–98.

Kistler, J. J. and M. Satyanarayanan (1992). Disconnected operation in the coda file system. *ACM Transactions on Computer Systems 10*(1), 3–25.

Klyne, G., F. Reynolds, and C. W. et al. (2003). Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies. Technical report, W3C Working Draft 25.

Kolodner, E. K., B. Liskov, and W. Weihl (1989, June). Atomic garbage collection: Managing a stable heap. *SIGMOD Record 18*(2), 15–25. Proc. of 1989 ACM SIGMOD Int'l Conf. on Management of Data.

Kolodner, E. K. and W. E. Weihl (1993, May). Atomic incremental garbage collection and recovery for large stable heap. In P. Buneman and S. Jajodia (Eds.), *Proc. of 1993 ASM SIGMOD Int'l Conf. on the Management of Data*, Washington, DC, pp. 177–186. Also MIT/LCS/TR-534, February, 1992.

Kordale, R. and M. Ahamad (1993, October). A scalable cyclic garbage detection algorithm for dist. systems. In E. Moss, P. R. Wilson, and B. Zorn (Eds.), *OOPSLA/ECOOP '93 W'shop on Garbage Collection in Object-Oriented Systems*.

Kordale, R., M. Ahamad, and M. Devarakonda (1996). Object caching in a CORBA-compliant system. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. USENIX.

Kordale, R., M. Ahamad, and J. Shilling (1993). Dist./concurrent garbage collection in dist. shared memory systems. In *Int'l W'shop on Object Orientation in Operating Systems*.

Ku, K.-I. and Y.-S. Kim (2000). Moflex transaction model for mobile heterogeneous multidatabase systems. In *Proceedings of the 10th International Workshop on Research Issues in Data Engineering*, San Diego, California.

Kubiatowicz, J., D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao (2000). Oceanstore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pp. 190–201. ACM Press.

Kumar, P. and M. Satyanarayanan (1993). Supporting application-specific resolution in an optimistically replicated file system. In *Workshop on Workstation Operating Systems*, pp. 66–70.

Ladin, R. and B. Liskov (1992, June). Garbage collection of a distributed heap. In *Int. Conf. on Distributed Computing Sys.*, Yokohama (Japan), pp. 708–715.

Lang, B., C. Quenniac, and J. Piquer (1992, January). Garbage collecting the world. In *Conf. Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pp. 39–50. ACM Press.

Lange, D. B. and M. Oshima (1998). *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley.

Lawrence, S., D. M. Pennock, G. W. Flake, R. Krovetz, F. M. Coetzee, E. Glover, F. A. Nielsen, A. Kruger, and C. L. Giles (2001, February). Persistence of web references in scientific research. *IEEE Computer vol 34(2)*.

Le Sergent, T. and B. Berthomieu (1992, September). Incremental multi-threaded garbage collection on virtually shared memory architectures. In *Proc. Int. Workshop on Memory Management*, Number 637 in Lecture Notes in Computer Science, Saint-Malo (France), pp. 179–199. Springer-Verlag.

Leach, P. and D. Perry (1996, nov). Cifs: A common internet file system. *Microsoft Internet Developer*.

Lecluse, C., P. Richard, and F. Velez (1988). O2, an object-oriented data model. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, New York, NY, USA, pp. 424–433. ACM Press.

Lemar, E. (2001, May). The design and evaluation of a storage system for pervasive computing. Technical report, Mew York University.

Lermen, C.-W. and D. Maurer (1986, August). A protocol for dist. reference counting. In *Conf. Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, Cambridge, MA, pp. 343–350. ACM Press.

Leser, N. (1992, November). The Distributed Computing Environment naming architecture. In *OpenForum*, Utrecht (NL), pp. 101–117.

Levanoni, Y. and E. Petrank (2001). An on-the-fly reference counting garbage collector for java. In *OOPSLA*, pp. 367–380.

Levanoni, Y. and E. Petrank (2006). An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst. 28*(1), 1–69.

Li, G. and H.-A. Jacobsen (2005). Composite subscriptions in content-based publish/subscribe systems. In G. Alonso (Ed.), *Middleware*, Volume 3790 of *Lecture Notes in Computer Science*, pp. 249–269. Springer.

Li, K. and P. Hudak (1986). Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, pp. 229–239. ACM Press.

Li, K. and P. Hudak (1989). Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems 7*(4), 321–359.

Lieberman, H. and C. E. Hewitt (1983). A real-time garbage collector based on the lifetimes of objects. *Comm. of the ACM 26(6)*, 419–29. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

Lindholm, T. and F. Yellin (1996). *The Java(TM) Virtual Machine Specification* (1st ed.). Addison-Wesley.

LinkAlarm (1998). Linkalarm, http://www.linkalarm.com/.

Lins, R. (2002a). Efficient cyclic weighted reference counting. In *14th Symposium on Computer Architecture and High Performance Computing (SCAB-PAD'02)*. IEEE.

Lins, R. (2005). A new multi-processor architecture for parallel lazy cyclic reference counting. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*. IEEE.

Lins, R. D. (1992a). Cyclic reference counting with lazy mark-scan. *Information Processing Letters 44*(4), 215–220. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

Lins, R. D. (1992b, September). Generational cyclic reference counting. Technical Report 22-92*, University of Kent, Canterbury, UK.

Lins, R. D. (2002b). An efficient algorithm for cyclic reference counting. *Inf. Process. Lett. 83*(3), 145–150.

Lins, R. D. and R. E. Jones (1993, May). Cyclic weighted reference counting. In K. Boyanov (Ed.), *Procedings of WP & DP'93 W'shop on Parallel and Dist. Processing*. North Holland. Also Computing Laboratory Technical Report 95, University of Kent, December 1991.

Lins, R. D. and M. A. Vasques (1991, August). A comparative study of algorithms for cyclic reference counting. Technical Report 92, Computing Laboratory, The University of Kent at Canterbury.

Liskov, B. (1989, April). A design a fault-tolerant, distributed implementation of Linda. Programming Methodology Group Memo 65, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA (USA).

Liskov, B., M. Castro, L. Shrira, and A. Adya (1999). Providing persistent objects in distributed systems. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, London, UK, pp. 230–257. Springer-Verlag.

Liskov, B., M. Day, and L. Shrira (1992, August). Distributed object management in Thor. In *Proc. Int. Workshop on Distributed Object Management*, Edmonton (Canada), pp. 1–15.

Liskov, B. and R. Ladin (1986, August). Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, Vancouver (Canada), pp. 29–39. ACM.

Lo, C., M. Chang, O. Frieder, and D. Grossman (2002). The object behavior of Java object-oriented database management systems. *Information Technology: Coding and Computing, 2002. Proceedings. International Conference on*, 247–252.

Louboutin, S. R. and V. Cahill (1997). Comprehensive dist. garbage collection by tracking causal dependencies of relevant mutator events. In *Proc. of ICDCS'97 Int'l Conf. on Dist. Computing Systems*. IEEE Press.

Lowry, M. C. (2004, dec). *A new approach to the train algorithm for distributed garbage collection.* Ph. D. thesis, Adelaide University, Australia.

Lowry, M. C. and D. S. Munro (2002). Safe and complete distributed garbage collection with the train algorithm. In *9th International Conference on Parallel and Distributed Systems (ICPADS 2002), 17-20 December 2002, Taiwan, ROC*, pp. 651–658.

Lu, Q. and M. Satyanarayanan (1995). Improving data consistency in mobile computing using isolation-only transactions. In *Fifth IEEE HotOS Topics Workshop*, Orcas Island, WA, USA.

Löwy, J. (2001). *COM and .NET Component Services*. O'Reilly Windows.

Maassen, J., T. Kielmann, and H. E. Bal (2000). Efficient replicated method invocation in java. In *Java Grande*, pp. 88–96.

Maheshwari, U. (1994, September). Fault-tolerant dist. garbage collection in a client-server object-oriented database. In *Third Int'l Conf. on Parallel and Dist. Information Systems, Austin*.

Maheshwari, U. and B. Liskov (1995). Collecting cyclic dist. garbage by controlled migration. In *Proc. of PODC'95 Principles of Dist. Computing*. Later appeared in Dist. Computing, Springer Verlag, 1996.

Maheshwari, U. and B. Liskov (1997a). Collecting cyclic dist. garbage by back tracing. In *Proc. of PODC'97 Principles of Dist. Computing*.

Maheshwari, U. and B. Liskov (1997b). Collecting cyclic distributed garbage by controlled migration. *Distributed Computing 10*(2), 79–86.

Maheshwari, U. and B. Liskov (1997c). Partitioned garbage collection of a large object store. In *Proc. of SIGMOD'97*.

Makpangou, M. and M. Shapiro (1988, March). The SOS object-oriented communication service. Rapport de Recherche 801, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France).

Malena, V. and M. Hapner (1999, dez). Enterprise javabeans(tm) specification 1.1 final release. Technical report, Sun Microsystems.

Mamei, M., F. Zambonelli, and L. Leonardi (2003). Tuples on the air: A middleware for context-aware computing in dynamic networks. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, DC, USA, pp. 342. IEEE Computer Society.

Maniatis, P., D. S. H. Rosenthal, M. Roussopoulos, M. Baker, T. Giuli, and Y. Muliadi (2003). Preserving peer replicas by rate-limited sampled voting. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, pp. 44–59. ACM Press.

Maniatis, P., M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker (2005). The lockss peer-to-peer digital preservation system. *ACM Trans. Comput. Syst. 23*(1), 2–50.

Marion, S. and R. Jones (2005). GCspy port to SSCLI (Rotor). presented at the 5th UK Memory Management Network Workshop.

Martinez, A. D., R. Wachenchauzer, and R. D. Lins (1990). Cyclic reference counting with local mark-scan. *Information Processing Letters 34*, 31–35.

Mascolo, C., L. Capra, and W. Emmerich (2001). An xml-based middleware for peer-to-peer computing. In *P2P '01: Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, pp. 69. IEEE Computer Society.

Mascolo, C., L. Capra, and W. Emmerich (2002a). Mobile computing middleware. pp. 20–58.

Mascolo, C., L. Capra, S. Zachariadis, and W. Emmerich (2002b). Xmiddle: A data-sharing middleware for mobile computing. *Wirel. Pers. Commun. 21*(1), 77–103.

Mattern, F. (1989). Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 215–226.

McBeth, J. H. (1963, September). On the reference counter method. *Comm. of the ACM 6*(9), 575.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. *Comm. of the ACM 3*, 184–195.

McFadden, T., K. Henricksen, J. Indulska, and P. Mascaro (2005). Applying a Disciplined Approach to the Development of a Context-Aware Communication Application. *3rd IEEE International Conference on Pervasive Computing and Communications (PerCom), IEEE Computer Society (2005)*, 300–306.

McKeown, M. (2003, jun). .net enterprise services and com+ 1.5 architecture. Technical report, Microsoft Corporation.

McLean, S., J. Naftel, and K. Williams (2002). *Microsoft .NET Remoting*. Microsoft Press.

McObject (2003). Perst - simple, fast, convenient object oriented database. http://www.perst.org.

Messer, A., I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu (2002). Towards a distributed platform for resource-constrained devices. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pp. 43. IEEE Computer Society.

Microsoft (1996, out). Activex data objects. Technical report, Microsoft.

Minsky, M. L. (1963, December). A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA.

Miranda, H., A. Pinto, and L. Rodrigues (2001). Appia, a flexible protocol kernel supporting multiple coordinated channels. *Proceedings of the 21st International Conference on Distributed Computing Systems*, 707–710.

Mocito, J., L. Rosa, N. Almeida, H. Miranda, L. Rodrigues, and A. Lopes (2006). Context adaptation of the communication stack. *International Journal of Parallel, Emergent and Distributed Systems 21*(3), 169–181.

Mohamed-Ali, K. A. (1984, December). *Object Oriented Storage Management and Garbage Collection in Dist. Processing Systems*. Ph. D. thesis, Royal Institute of Technology, Stockholm.

Monson-Haefel, R. (2000, mar). *Enterprise JavaBeans.* (2nd ed.). O'Reilly Press.

Montanari, R. and G. Tonti (2002, June). A policy-based infrastructure for the dynamic control of agent mobility. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, Monterrey (USA).

Moon, D. A. (1984, August). Garbage collection in a large LISP system. See Steele (1984), pp. 235–245.

Moreau, L. (1998a, September). A Distributed Garbage Collector with Diffusion Tree Reorganisation and Object Mobility. In *Proc. of the Third Int'l Conf. of Functional Programming (ICFP'98)*.

Moreau, L. (1998b, October). Hierarchical Distributed Reference Counting. In *Proc. of the First Int'l Symposium on Memory Management (ISMM'98)*.

Moreau, L. (2001). Tree rerooting in distributed garbage collection: Implementation and performance evaluation. *Higher Order Symbol. Comput. 14*(4), 357–386.

Moreau, L., P. Dickman, and R. Jones (2003, July). Birrell's distributed reference listing revisited. Technical Report 8–03, University of Kent, Canterbury. (later accepted for publication in ACM Transactions On Programming Languages And Systems as vol.27(6), 2005).

Moreau, L., P. Dickman, and R. Jones (2005). Birrell's distributed reference listing revisited. *ACM Trans. Program. Lang. Syst. 27*(6), 1344–1395.

Moreau, L. and N. Gray (1998). A community of agents maintaining link integrity in the world wide web. In *Proceedings of the 3rd International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-98)*, London, UK.

Moss, J. E. B. (1989, June). Addressing large distributed collections of persistent objects: The Mneme project's approach. In *Proceedings of the Second International Workshop on Database Programming Languages*, Gleneden Beach, OR (USA). Also available as COINS Technical Report 89–68, Object-Oriented Systems Laboratory, Dept. of Computer and Information Science, University of Massachusetts, Amherst.

Moss, J. E. B., D. S. Munro, and R. L. Hudson (1996, May). PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proc. of the 6th Int. Workshop on Persistent Object Systems*, Cape May NJ (USA).

Mueller, J., D. Polansky, P. Novak, C. Foltin, and D. Polivaev (2003). Freemind - free mind mapping software. http://freemind.sourceforge.net/wiki/index.php/Main_Page.

Munro, D. S. and A. L. Brown (2001). Evaluating partition selection policies using the pmos garbage collector. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems*, London, UK, pp. 125–137. Springer-Verlag.

Murphy, A., G. Picco, and G. Roman (2001). Lime: A Middleware for Physical and Logical Mobility. *Proceedings of the 21 stInternational Conference on Distributed Computing Systems*, 524–533.

Muthukumar, R. and D. Janakiram (2006, feb). Yama: A scalable generational garbage collector for java in multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems 17*(2), 148–159.

Narasimhan, N., L. Moser, and P. Melliar-Smith. Transparent Consistent Replication of Java RMI Objects. *Second International Symposium, Distributed Objects & Applications (DOA 2000)*, 17–26.

Narasimhan, N., L. Moser, and P. Melliar-Smith (2001). Interceptors for Java Remote Method Invocation. *Concurrency and Computation Practice and Experience 13*(8-9), 755–774.

Nettles, S., J. O'Toole, D. Pierce, and N. Haines (1992, September). Replication-based incremental copying collection. In *Proc. Int. Workshop on Memory Management*, Number 637 in Lecture Notes in Computer Science, Saint-Malo (France), pp. 357–364. Springer-Verlag.

Network Working Group (1989). Network file system, request for comments: 1094.

Network Working Group (1996). Hypertext transfer protocol, request for comments: 1945.

Nicoloudis, N. and D. Pratistha (2003, jul). .net compact framework mobile web server architecture. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetcomp/html/NETCFMA.asp, Monash University, Caulfield, Australia & MSDN, Microsoft.

Niederst, J. (1999). *HTML Pocket Reference* (1 ed.). OReilly.

Nilsson, M., J. Hjelm, and H. Ohto (2000). Composite Capabilities/Preference Profiles: Requirements and Architecture. Technical report, W3C Working Draft 21.

Noble, B., M. Satyanarayanan, and M. Price (1995). A programming interface for application-aware adaptation in mobile computing. In *MLICS '95: Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, Berkeley, CA, USA, pp. 57–66. USENIX Association.

Noble, B. D., M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker (1997). Agile application-aware adaptation for mobility. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, New York, NY, USA, pp. 276–287. ACM Press.

Norcross, S., R. Morrison, D. S. Munro, H. Detmold, and K. E. Falkner (2005). Implementing a family of distributed garbage collectors. *Journal of Research and Practice in Information Technology 37*(1).

Nori, A. K. (1979). A storage reclamation system for an applicative multiprocessor system. Master's thesis, University of Utah, Salt Lake City, Utah.

Novell (2004). The mono project. http://www.mono-project.com/.

Nunn, M. (2005, jun). An overview of sql server 2005 for the database developer. Technical report, Microsoft.

Object Management Group (2002). Corba component model (ccm). Technical report, Object Management Group, Inc.

ObjectSpace, I. (1997, September). Objectspace voyager core technology. Objectspace technical report, ObjectSpace, Inc.

O'Neill, E. T., B. F. Lavoie, and R. Bennett (2003). Trends in the evolution of the public web 1998 - 2002. *D-Lib Magazine 9*(4).

O'Toole, J., S. Nettles, and D. Gifford (1993, December). Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC (USA), pp. 161–174.

Pairot, C., P. García, and A. F. G. Skarmeta (2004). Dermi: A decentralized peer-to-peer event-based object middleware. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 236–243. IEEE Computer Society.

Pallickara, S. and G. Fox (2003). NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. *Proceedings of the International Middleware Conference*.

Parlavantzas, N., G. Coulson, and G. Blair (2003, june). A resource adaptation framework for reflective middleware. In *The 2nd International Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil.

Paterson, J., S. Edlich, H. Hörning, and R. Hörning (2006). The Definitive Guide to db4o.

Patterson, L. I., R. S. Turner, and R. M. Hyatt (1993). Construction of a fault-tolerant distributed tuple-space. In *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pp. 279–285. ACM Press.

Pereira, P., L. Veiga, and P. Ferreira (2006, october). Extending .net remoting with distributed garbage collection. In *2nd International Conference on Innovative Views for .Net Technologies (IV.Net 2006)*, Lecture Notes in Computer Science, Brazil. Springer-Verlag.

Petersen, K., M. Spreitzer, D. Terry, M. Theimer, and A. Demers (1997, December). Flexible update propagation for weakly consistent replication. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo (France), pp. 288–301. ACM.

Petrovic, M., I. Burcea, and H.-A. Jacobsen (2003). S-topss: Semantic toronto publish/subscribe system. In *VLDB*, pp. 1101–1104.

Philippsen, M. (2000, May). Cooperating distributed garbage collectors for clusters and beyond. *Concurrency: Practice and Experience 12*(7), 595–610.

Piquer, J. M. (1991, June). Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, Volume 505 of *Lecture Notes in Computer Science*, Eindhoven (the Netherlands), pp. 150–165. Springer-Verlag.

Piquer, J. M. (1996, September). Indirect dist. garbage collection: Handling object migration. *ACM Trans. on Programming Languages and Systems 18*(5), 615–647.

Pitoura, E. and B. K. Bhargava (1999). Data consistency in intermittently connected distributed systems. *Knowledge and Data Engineering 11*(6), 896–915.

Plainfossé, D. and M. Shapiro (1995, September). A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK).

Platt, D. (1999). *Understanding COM+*. Microsoft Press.

Platt, D. S. (2001). *Introducing Microsoft .Net*. Microsoft Press. ISBN: 0-7356-1377-X.

Popek, G. J., R. G. Guy, T. W. Page, Jr., and J. S. Heidemann (1990, November). Replication in Ficus distributed file systems. In *Proceedings of the Workshop on Management of Replicated Data*, pp. 20–25. University of California, Los Angeles: IEEE.

Prasad, S. K., V. K. Madisetti, S. B. Navathe, R. Sunderraman, E. Dogdu, A. G. Bourgeois, M. Weeks, B. Liu, J. Balasooriya, A. Hariharan, W. Xie, P. Madiraju, S. Malladi, R. Sivakumar, A. Zelikovsky, Y. Zhang, Y. Pan, and S. Belkasim (2004, oct). Syd: A middleware testbed for collaborative applications over small heterogeneous devices and data stores. In *ACM/IFIP/USENIX International Middleware Conference*, Toronto, Canada, pp. 352–371.

Preguiça, N., C. Baquero, J. L. Martins, M. Shapiro, P. S. Almeida, H. Domingos, V. Fonte, and S. Duarte (2005). Few: File management for portable devices. In *International Workshop on Software Support for Portable Storage*.

Preguiça, N., C. Baquero, F. Moura, J. L. Martins, R. Oliveira, H. J. L. Domingos, J. O. Pereira, and S. Duarte (2000). Mobile transaction management in mobisnap. In *ADBIS-DASFAA*, pp. 379–386.

Preguiça, N., J. L. Martins, M. Cunha, and H. Domingos (2003). Reservations for conflict avoidance in a mobile database system. In *Proc. of the 1st Usenix Int'l Conference on Mobile Systems, Applications and Services (Mobisys 2003)*.

Preguiça, N., M. Shapiro, and J. L. Martins (2003a). Sqlicecube: Automatic sematics-based reconciliation for mobile databases. Technical report, DI-FCT-UNL TR-02-2003.

Preguiça, N. M., J. L. Martins, S. Duarte, and H. J. L. Domingos (2001). Supporting disconnected operation in doors. In *Proceedings of HotOS-VIII: 8th Workshop on Hot Topics in Operating Systems, May 20-23, 2001, Elmau/Oberbayern, Germany*, pp. 179.

Preguiça, N. M., M. Shapiro, and C. Matheson (2003b). Semantics-based reconciliation for collaborative and mobile environments. In *CoopIS/DOA/ODBASE*, pp. 38–55.

Printezis, T. and A. Garthwaite (2002). Visualising the train garbage collector. In *MSP/ISMM*, pp. 157–170.

Printezis, T. and R. Jones (2002). *GCspy: an adaptable heap visualisation framework*. ACM Press New York, NY, USA.

Queinnec, C., B. Beaudoing, and J.-P. Queille (1989). Mark DURING Sweep rather than Mark THEN Sweep. *Lecture Notes in Computer Science 365*, 224–237.

R. Srinivasan, Network Working Group, S. M. I. (1995). Rpc: Remote procedure call protocol specification version 2, request for comments: 1831.

Ramamritham, K. and P. K. Chrysanthis (1996). A taxonomy of correctness criterion in database applications. *Journal of Very Large Databases 4*(1).

Rana, S. P. (1983, July). A dist. solution to the dist. termination problem. *Information Processing Letters 17*, 43–46.

Ratner, D., P. Reiher, and G. J. Popek (2004). Roam: a scalable replication system for mobility. *Mob. Netw. Appl. 9*(5), 537–544.

Ratner, D., P. Reiher, G. J. Popek, and G. H. Kuenning (2001). Replication requirements in mobile environments. *Mob. Netw. Appl. 6*(6), 525–533.

Reich, V. and D. Rosenthal (2001, June). Lockss: A permanent web publishing and access system. *D-Lib M'zine 7*.

Reingold, E. M. (1973, May). A non-recursive list moving algorithm. *Comm. of the ACM 16*(5), 305–307.

Rhea, S., P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz (2003). Pond: The oceanstore prototype. In *Proceedings of the Conference on File and Storage Technologies*. USENIX.

Richer, N. and M. Shapiro (2000). The memory behavior of the WWW, or the WWW considered as a persistent store. In *POS 2000*, pp. 161–176.

Rodrigues, H. and R. Jones (1998). Cyclic distributed garbage collection with group merger. *Lecture Notes in Computer Science 1445*, 260.

Rodrigues, H. C. C. D. and R. E. Jones (1996, October). A cyclic dist. garbage collector for Network Objects. In O. Babaoglu and K. Marzullo (Eds.), *Tenth Int'l W'shop on Dist. Algorithms WDAG'96*, Volume 1151 of *Lecture Notes in Computer Science*, Bologna. Springer-Verlag.

Rodrigues da Silva, A., A. Romão, D. Deugo, and M. M. da Silva (2001). Towards a reference model for surveying mobile agent systems. *Autonomous Agents and Multi-Agent Systems 4*(3), 187–231.

Rodriguez, P., C. Spanner, and E. Biersack (2001). Analysis of web caching architectures: Hierarchical and distributed caching.

Rodriguez-Rivera, G. and V. Russo (1997). Cyclic distributed garbage collection without global synchronization in corba. In *OOPSLA'97 GC & MM Workshop*.

Roman, M., C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt (2002a). Gaia: a middleware platform for active spaces. *SIGMOBILE Mob. Comput. Commun. Rev. 6*(4), 65–67.

Roman, M., C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt (2002b). A middleware infrastructure for active spaces. *IEEE Pervasive Computing 1*(4), 74–83.

Rosenthal, D. and V. Reich (2000, June). Permanent web publishing. In *Freenix Track, Usenix Annual Technical Conference, Usenix*, Berkeley, California.

Rudalics, M. (1990). Correctness of distributed garbage collection algorithms. Technical Report 90-40.0, Johannes Kepler Universitat, Linz Austria.

Russel, C. (2002, apr). Javatm data objects specification (jsr 12): Final release. Technical report, Sun Microsystems.

Russel, C. (2003, may). Javatm data objects specification (jsr 12): Final release 2. Technical report, Sun Microsystems.

Russel, C. (2005, aug). Javatm data objects 2 specification (jsr 243): Proposed final draft. Technical report, Sun Microsystems.

Russel, C. (2006, feb). Javatm data objects 2 specification (jsr 243): Final release. Technical report, Sun Microsystems.

S. Shepler, e. a. (2003). Network file system, request for comments: 3530.

Saito, Y. and M. Shapiro (2005). Optimistic replication. *ACM Comput. Surv. 37*(1), 42–81.

Salber, D., A. K. Dey, and G. D. Abowd (1999). The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 434–441. ACM Press.

Samples, A. D. (1992, 16–18 September). Garbage collection-cooperative C++. See Bekkers and Cohen (1992).

Sanchez, A., L. Veiga, and P. Ferreira (2001, January). Distributed garbage collection for wide area replicated memory. In *Proc. of the Sixth USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio (USA).

Santos, N., L. Veiga, and P. Ferreira (2004). Transaction policies for mobile networks. In *5th IEEE International Workshop on Policies for Dist. Systems and Networks(Policy 2004).*

Satyanarayanan, M. (1996). Fundamental challenges in mobile computing. In *Symposium on Principles of Distributed Computing*, pp. 1–7.

Satyanarayanan, M. (2002). The evolution of coda. *ACM Trans. Comput. Syst. 20*(2), 85–124.

Schilit, B., N. Adams, and R. Want (1994). Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US.

Serrano-Alvarado, P., C. Roncancio, and M. Adiba (2004). A survey of mobile transactions. *Distrib. Parallel Databases 16*(2), 193–230.

Sessions, R. (1998, December). *COM and DCOM: Microsoft's Vision for Dist. Objects*. Wiley. ISBN 0-471-19381-X.

Shannon, B. (2001, aug). Java 2 platform, enterprise edition 1.3 specification (jsr 58): Final release. Technical report, Sun Microsystems.

Shannon, B. (2003, nov). Java 2 platform, enterprise edition 1.4 (j2ee 1.4) specification (jsr 151): Final release. Technical report, Sun Microsystems.

Shannon, B. (2006, may). Java platform, enterprise edition 1.5 (jee 1.5) specification (jsr 244): Final release. Technical report, Sun Microsystems.

Shapiro, M. (1986, May). Structure and encapsulation in distributed systems: the proxy principle. In *Proc. of the 6th Intl. Conf. on Dist. Systems*, Boston, pp. 198–204.

Shapiro, M. (1991a, September). A fault-tolerant, scalable, low-overhead dist. garbage collection protocol. In *Proc. of the Tenth Symposium on Reliable Dist. Systems*, Pisa.

Shapiro, M. (1991b, July). Soul: An object-oriented OS framework for object support. In *Workshop on Operating Systems for the Nineties and Beyond*, Dagstuhl Castle, Germany, pp. 251–255. Springer-Verlag.

Shapiro, M., P. Dickman, and D. Plainfossé (1992a, August). Robust, dist. references and acyclic garbage collection. In *Symposium on Principles of Dist. Computing*, Vancouver, Canada.

Shapiro, M., P. Dickman, and D. Plainfossé (1992b, November). SSP chains: Robust, dist. references supporting acyclic garbage collection. Rapports de Recherche 1799, Institut National de la Recherche en Informatique et Automatique. Also available as Broadcast Technical Report 1.

Shapiro, M., F. L. Fessant, and P. Ferreira (2000). Recent advances in distributed garbage collection. *Lecture Notes in Computer Science 1752*, 104.

Shapiro, M., O. Gruber, and D. Plainfossé (1990, November). A garbage detection protocol for a realistic dist. object-support system. Rapports de Recherche 1320, INRIA-Rocquencourt. Superseded by (Shapiro 1991a).

Shapiro, M., N. Preguiça, and J. O'Brien (2004, jan). Rufis: mobile data sharing using a generic constraint-oriented reconciler. In *IEEE International Conference on Mobile Data Management (MDM 2004)*.

Siegel, J. (1996). *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc.

Sinha, A. (1992). Client-server computing. *Commun. ACM 35*(7), 77–98.

Skubiszewski, M. and N. Porteix (1996, April). GC-consistent cuts of databases. Rapport de recherche 2681, Institut National de la Recherche en Informatique et Automatique, rocquencourt. ftp://ftp.inria.fr/INRIA/publication/RR/RR-2681.ps.gz.

Skubiszewski, M. and P. Valduriez (1997). Concurrent garbage collection in O2. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld (Eds.), *Proc. of 23rd Int'l Conf. on Very Large Databases*, Athens, pp. 356–365. Morgan Kaufman.

Sousa, P., M. Sequeira, A. Zúquete, P. Ferreira, C. Lopes, J. Pereira, P. Guedes, and J. Marques (1993, nov). Distribution and persistence in the ik platform. *Computing Systems Journal 6(4)*.

Steele, G. L. (1975, September). Multiprocessing compactifying garbage collection. *Comm. of the ACM 18*(9), 495–508.

Steele, G. L. (1976, June). Corrigendum: Multiprocessing compactifying garbage collection. *Comm. of the ACM 19*(6), 354.

Steele, G. L. (Ed.) (1984, August). *Conf. Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas. ACM Press.

Stutz, D. (2002, march). The microsoft shared source cli implementation. MSDN Library Article, Microsoft Corporation.

Sun Microsystems (2001). Sun java bug database, sun developer network. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4403367.

Sun Microsystems (2004). Java rmi documentation.
http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html.

Sundell, H. (2005). Wait-free reference counting and memory management. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), 4-8 April 2005, Denver, CA, USA*.

Swaminathan, N. and S. Raghavan (2000, August). Intelligent prefetch in www using client behavior characterization. In *8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 13–19.

Taivalsaari, A. (2000, may). J2me connected, limited device configuration (cldc) version 1.0 (jsr 30): Final release. Technical report, Sun Microsystems.

Taivalsaari, A. (2003, march). J2me connected, limited configuration (cldc) version 1.1 (jsr 139): Final release. Technical report, Sun Microsystems.

Talia, D. and P. Trunfio (2003). Toward a Synergy Between P2P and Grids. *IEEE Internet Computing 7*(04), 96–95.

Tam, D., R. Azimi, and H. Jacobsen (2003). Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables. *Proceedings of International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, 138–152.

Tel, G. and F. Mattern (1993, January). The derivation of dist. termination detection algorithms from garbage collection schemes. *ACM Trans. on Programming Languages and Systems 15*(1).

Tel, G., R. B. Tan, and J. van Leeuwen (1987). The derivation of on-the-fly garbage collection algorithms from dist. termination detection protocols. *Lecture Notes in Computer Science 247*, 445–455.

Tennison, J. (2001). *XSLT and XPath On The Edge* (1 ed.). Wiley.

Terry, D. B., K. Petersen, M. Spreitzer, and M. Theimer (1998). The case for non-transparent replication: Examples from bayou. *IEEE Data Eng. Bull. 21*(4), 12–20.

Terry, D. B., M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser (1995). Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 172–182. ACM Press.

Thompson, D. and C. Miller (2006, sep). Microsoft's.net microframework, product positioning and technology whitepaper. http://www.aboutnetmf.com/entry.asp.

Udell, J. (1999). Exploring XML-RPC. *Byte Magazine, August*.

Ungar, D. (1984). Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not. 19*(5), 157–167.

Ungar, D. M. and F. Jackson (1988). Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices 23*(11), 1–17.

van Reeuwijk, C. and H. Sips (2005). Adding tuples to Java: a study in lightweight data structures. *Concurrency and Computation Practice and Experience 17*(5-6), 423–438.

van Reeuwijk, C. and H. J. Sips (2002). Adding tuples to java: a study in lightweight data structures. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pp. 185–191. ACM Press.

Vardhan, A. and G. Agha (2002). Using passive object garbage collection algorithms for garbage collection of active objects. In *MSP/ISMM*, pp. 213–220.

Veiga, L. (2002). Incremental replication in obiwan. Crash course v2 for faculty and phds, available at:
http://research.microsoft.com/collaboration/university/europe/Events/dotnetcc/Version2/DVD/, Microsoft Research Cambridge.

Veiga, L. and P. Ferreira (2001, Nov). Mobility and wireless support in OBIWAN. In *3rd IFIP/ACM Middleware Conference*, Heidelberg (Germany).

Veiga, L. and P. Ferreira (2002a, July). Incremental replication for mobility support in OBIWAN. In *The 22nd International Conference on Distributed Computer Systems*, Viena (Austria), pp. 249–256.

Veiga, L. and P. Ferreira (2002b, Sep). Mobility support in OBIWAN. In *2nd Microsoft Research Summer Workshop*, Cambridge (UK).

Veiga, L. and P. Ferreira (2003a, oct). Complete distributed garbage collection, an experience with rotor. *IEE Research Journals - Software 150(5).*

Veiga, L. and P. Ferreira (2003b, sep). Complete distributed garbage collection, an experience with rotor. In *2nd Microsoft Research Annual Workshop on the Shared Source Common Language Infrastructure (Rotor CLR)*, Redmond, WA, USA.

Veiga, L. and P. Ferreira (2003c, Mar). Repweb: Replicated web with referential integrity. In *18th ACM Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA.

Veiga, L. and P. Ferreira (2004a, October). Poliper : Policies for mobile and pervasive environments. In *3rd Workshop on Reflective and Adaptive Middleware. In 6th ACM International Middleware Conference*, Toronto, Canada.

Veiga, L. and P. Ferreira (2004b, April). Turning the web into an effective knowledge repository. In *6th International Conference on Enterprise Information Systems*, Porto, Portugal, pp. 154–161.

Veiga, L. and P. Ferreira (2005a, april). Asynchronous complete distributed garbage collection. In *19th IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, USA.

Veiga, L. and P. Ferreira (2005b, april). A comprehensive approach for memory management of replicated objects. Technical report rt/07/2005, INESC-ID Lisboa.

Veiga, L. and P. Ferreira (2005c, september). Transparent object-swapping for resource-constrained devices. Technical report rt/xx/2005, INESC-ID Lisboa.

Veiga, L., N. Santos, R. Lebre, and P. Ferreira (2004). Loosely-coupled, mobile replication of objects with transactions. In *Workshop on Qos and Dynamic Systems. 10th IEEE International Conference On Parallel and Distributed Systems(ICPADS 2004).*

Vestal, S. C. (1987). *Garbage collection: an exercise in distributed, fault-tolerant programming*. Ph. D. thesis, Seattle, WA, USA.

W3C-HTML. Hypertext markup language (html). Technical report.

W3C-XML. Extensible markup language (xml). Technical report.

Walborn, G. D. and P. K. Chrysanthis (1995). Supporting semantics-based transaction processing in mobile database applications. In *Symposium on Reliable Distributed Systems*, pp. 31–40.

Wang, J. (1999). A survey of web caching schemes for the internet. *SIGCOMM Comput. Commun. Rev. 29*(5), 36–46.

Wang, W. and C. A. Varela (2006, May). Distributed garbage collection for mobile actor systems: The pseudo root approach. In *Proceedings of the First International Conference on Grid and Pervasive Computing (GPC 2006)*, Taichung, Taiwan. Springer-Verlag.

Watson, P. and I. Watson (1987, June). An efficient garbage collection scheme for parallel computer architectures. In *PARLE'87—Parallel Architectures and Languages Europe*, Number 259 in Lecture Notes in Computer Science, Eindhoven (the Netherlands). Springer-Verlag.

Weiser, M. (1991). The computer for the twenty-first century. *Scientific American* (9).

Weiser, M. (1993). Some computer science problems in ubiquitous computing. *Communications of the ACM 36*(7), 75–84.

Weiser, M., A. Demers, and C. Hauser (1989, December). The Portable Common Runtime approach to interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park AZ (USA), pp. 114–122. ACM.

Weizenbaum, J. (1962). Knotted list structures. *Comm. of the ACM 5*(3), 161–165.

Wentworth, E. P. (1990, July). Pitfalls of conservative garbage collection. *Software—Practice and Experience 20*, 719–727.

White, S. J. and D. J. Dewitt (1992). A performance study of alternative object faulting and pointer swizzling strategies. In *18th VLDB Conf. Vancouver, British Columbia, Canada*.

Wigley, A., S. Wheelwright, R. Burbidge, R. MacLeod, and M. Sutton (2003). *Microsoft .NET Compact Framework (Core Reference)*. Microsoft Press.

Willard, B. and O. Frieder (1998). Autonomous garbage collection: Resolving memory leaks in long running network applications. In *ICCCN*, pp. 886–896.

Willard, B. and O. Frieder (2000). Autonomous garbage collection: resolving memory leaks in long-running server applications. *Computer Communications 23*(10), 887–900.

Wilson, P. (1991). Operating system support for small objects. *Object Orientation in Operating Systems, 1991. Proceedings., 1991 International Workshop on*, 80–86.

Wilson, P. (1996, march). Distributed garbage collection general discussion for faq. GCList Mailing List (gclist@iecc.com).

Wilson, P. R. (1990, December). Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. Technical Report UIC–EECS–90–6, University of Illinois at Chicago, Electrical Engineering and Computer Science Department, Chicago, Illinois. Also in Computer Architecture News, 19(4):6–13, June 1991.

Wilson, P. R. (1992, September). Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Number 637 in Lecture Notes in Computer Science, Saint-Malo (France). Springer-Verlag. ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps.

Wilson, P. R. and S. V. Kakkad (1992, September). Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Int'l W'shop On Object Orientationin Operating Systems. Paris, France, 364-377*.

Winer, D. (1999). XML-RPC Specification. *URL: http://www. xmlrpc. com/-spec.*

Wise, D. S. and D. P. Friedman (1977). The one-bit reference count. *BIT 17*(3), 351–9.

Wollrath, A., R. Riggs, and J. Waldo (1996, June). A distributed object model for the Java system. In *Conf. on Object-Oriented Technologies*, Toronto, Ontario (Canada). Usenix.

www.fipa.org (2002). Foundation of intelligent physical agents.

Wyckoff, P. (1998). Tspaces. *IBM Systems Journal 37*(3).

XenuLink (1997). Linksleuth http://home.snafu.de/tilman/.

Yu, W. and A. Cox (1996, May). Conservative garbage collection on dist. shared memory systems. In *The 16th Int'l Conf. on Dist. Computer Systems*, Hong-Kong, pp. 402–410.

Yuasa, T. (1990). Real-time garbage collection on general-purpose machines. *Journal of Software and Systems 11*(3), 181–198.

Zdonik, S. and D. Maier (1990). *Readings in Object-Oriented Database Systems*. San Mateo, California (USA): Morgan-Kaufman.

Zhou, D., N. Islam, and A. Ismael (2004). Flexible on-device service object replication with replets. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pp. 131–142. ACM Press.

Zorn, B. (1990a, June). Comparing mark-and-sweep and stop-and-copy garbage collection. In *Conf. Record of the 1990 ACM Symposium on Lisp and Functional Programming*, Nice, France. ACM Press.

Zorn, B. (1990b, October). Designing systems for evaluation: A case study of garbage collection. See Jul and Juul (1990).

Zorn, B. (1993). The measured cost of conservative garbage collection. *Software Practice and Experience 23*, 733–756.

Zorn, B. G. (1989, March). *Comparative Performance Evaluation of Garbage Collection Algorithms*. Ph. D. thesis, University of California at Berkeley. Technical Report UCB/CSD 89/544.