OBIWAN: Design and Implementation of a Middleware Platform

Paulo Ferreira, *Member*, *IEEE*, Luís Veiga, and Carlos Ribeiro

Abstract—Programming distributed applications supporting data sharing is very hard. In most middleware platforms, programmers must deal with system-level issues for which they do not have the adequate knowledge, e.g., object replication, abusive resource consumption by mobile agents, and distributed garbage collection. As a result, programmers are diverted from their main task: the application logic. In addition, given that such system-level issues are extremely error-prone, programmers spend innumerous hours debugging. We designed, implemented, and evaluated a middleware platform called OBIWAN that releases the programmer from the above mentioned system-level issues. OBIWAN has the following distinctive characteristics: 1) allows the programmer to develop applications using either remote object invocation, object replication, or mobile agents, according to the specific needs of applications, 2) supports automatic object replication (e.g., incremental on-demand replication, transparent object faulting and serving, etc.), 3) supports distributed garbage collection of useless replicas, and 4) supports the specification and enforcement of history-based security policies well adapted to mobile agents needs (e.g., preventing abusive resource consumption).

Index Terms—Middleware, replication, distributed garbage collection, security policies, mobile agents.

1 Introduction

The need for sharing is well-known in a large number of distributed applications. These applications are difficult to develop for wide area (possibly mobile) networks because of slow and unreliable connections and, most of all, because programmers are forced to deal with system-level issues (e.g., distributed garbage collection, replication, etc.).

In addition, 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. For example, there are 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. There are also situations in which an application would be preferably developed using mobile agents instead of traditional remote object invocation (RMI).

As an example, consider a shopping (client) program running in a PDA. The client interacts with servers to get information about the products in their catalogs. However, catalogs are too big to fit in the PDA's memory and, most importantly, the client program is aware of the shopping profile and needs of its owner. In addition, as the user browses the existing products related to his needs, lots of invocations take place. Thus, even while connected to the network (via GPRS/craddle/etc.) it is advantageous (in terms of performance and cost) to replicate only those products whose category the user is interested in. Even when the PDA is not connected, the user can still fill his shopping cart as most of the data he needs (according to his profile) is replicated in his PDA. Once the user has filled his shopping cart, the buying phase may start; this implies accessing different shopping servers in the network (so that products are bought at the most advantageous server). Thus, the client program opens a connection and sends a mobile agent to the network; then, the connection is closed. The agent interacts with the shopping servers as needed. Obviously, this interaction has to be controlled according to some security policy. Later, the client program opens a network connection and invokes the agent via RMI in order to provide final payment authorization and delivery details.

Currently, dealing with such programming paradigm diversity implies: 1) either using different middleware platforms, with obvious inconveniencies such as integration problems, security loopholes, and learning costs, or 2) when using a single middleware platform, 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, among other details.

For these reasons, we designed, implemented, and evaluated a platform called OBIWAN,[1] which has the following distinctive characteristics:

1. **Paradigm Flexibility:** allows programmers to develop applications using either RMI, object replication, or mobile agents,[2] according to the specific needs of applications.
2. **Automatic Replication:** supports distributed memory management capable of dealing with object replicas automatically (e.g., incremental on-demand replication, transparent object faulting and serving, etc.).
3. **Distributed Garbage Collection (DGC):** supports the automatic reclamation of useless replicas.
4. **Security Policies:** supports the definition and enforcement of history-based security policies well

• *The authors are with INESC ID/IST, Rua Alves Redol No. 9-6 Andar, 1000-029 Lisboa, Portugal.*
E-mail: {paulo.ferreira, luis.veiga, carlos.ribeiro}@inesc-id.pt.

Manuscript received 8 Dec. 2002; revised 3 July 2003; accepted 3 Aug. 2003. For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 118747.

1. OBIWAN stands for **O**bject **B**roker **I**nfrastructure for **W**ide **A**rea **N**etworks.

2. We will hereafter refer to mobile agent, or simply agent, as a program that travels across a network, possibly acting autonomously and on behalf of his owner.

adapted to agents needs (e.g., preventing abusive resource consumption, enforcing a Chinese-wall policy, etc.).

No other middleware platform provides all the characteristics mentioned above. In particular, the support for replication raises the problem of DGC more seriously than with traditional RMI; garbage collection algorithms for distributed systems based on RMI are not safe when applied to a distributed system with replicas [10]. In addition, it is equally important to note that the use of mobile agents brings the problem of abusive resource consumption on hosting computers. OBIWAN solves this problem by means of a security language and a monitor that enforces the security policies thus defined (not necessarily by the programmer, i.e., they can be defined by the policy administrator).

Finally, OBIWAN runs both on top of Java [1] and .Net [25] and does not require any modification of the underlying Java Virtual Machine (JVM) or Common Language Runtime (CLR), respectively.

1.1 Replication

The replication module in OBIWAN is responsible for dealing with all aspects of replica creation so that: 1) it allows the application to decide, at runtime, the mechanism by which objects should be invoked, either via RMI or invocation on a local replica, 2) it allows incremental replication of large object graphs, and 3) it allows the creation of dynamic clusters of objects. These mechanisms allow an application to deal with situations that frequently occur in a mobile or wide-area network, such as disconnections and slow links: 1) as long as objects needed by an application (or by an mobile agent) are colocated, there is no need to be connected to the network, and 2) it is possible to replace, at runtime, remote by local invocations on replicas, thus improving the performance and adaptability of applications.

1.2 Distributed Garbage Collection

Concerning DGC, most algorithms [24] are not well-suited for systems supporting object replication because: either 1) they do not consider the existence of replication or 2) they impose severe constraints on scalability by requiring causal delivery to be provided by the underlying communication layer. (More details are given in Section 2.2).

In OBIWAN, the DGC algorithm solves both these problems. The result is an algorithm that, besides being correct in the presence of replicated objects and independent of the protocol that maintains such replicas coherent among processes, it does not require causal delivery to be ensured by the underlying communications support. In addition, it has minimal performance impact on applications.

1.3 Mobile Agents Security

The mobile agent paradigm [16], [36] has introduced some new concerns on the security area, in particular, in information flow control and authorization. A major issue with such agent-based applications is the definition of what operations the agent should be authorized to perform and what operations the agent should be prohibited from doing or obliged to do. This can be accomplished with the use of history-based security policies [8], [28].

OBIWAN supports the definition and enforcement of history-based security policies. The support for these policies is extremely important in order to implement real organization security policies, where an agent's behavior influences its permissions; we can define some useful and complex history-based security policies applied to the mobile agent paradigm, such as Chinese-wall [3] and history-based separation of duty [33]. Thus, an agent's host is able to allow or deny an agent's request to access a protected resource based on the agent's past behavior, whether on that host or on previous hosts.

This paper is organized as follows: In Section 2, we describe the architecture of OBIWAN focusing on its most relevant components: support for incremental replication, DGC, and mobile agents security. Section 3 presents the most important aspects of the implementation. Then, in Section 4, we show some relevant performance results. In Sections 5 and 6, we compare our work with others' and draw some conclusions, respectively.

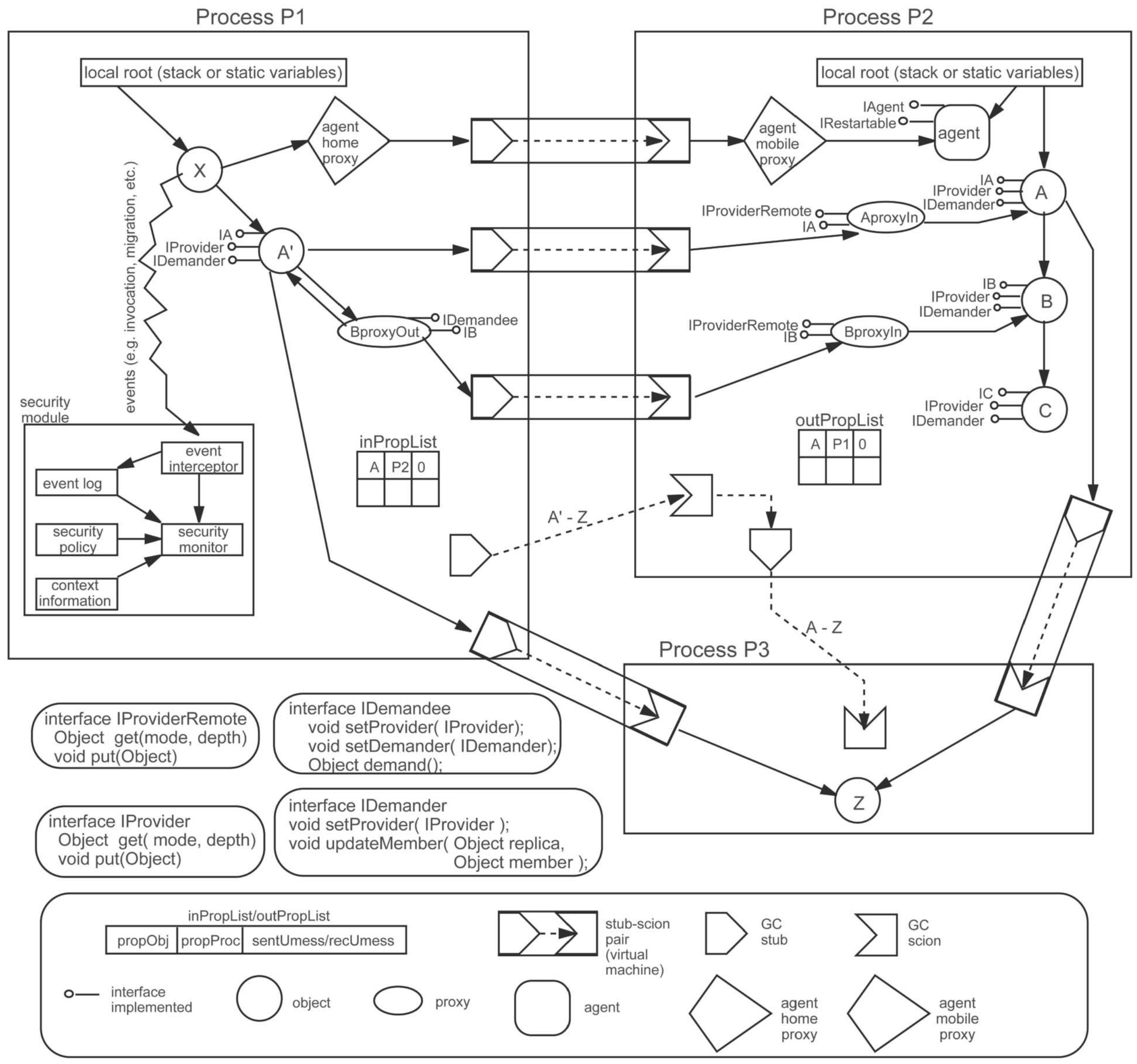
2 Architecture

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. Thus, with regard to replication, 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).

OBIWAN gives to the application programmer the view of a network of computers in which one or more processes run; objects and agents exist inside processes (with regard to agents, we call such processes hosts). An object can be invoked locally (after being replicated) or remotely. Mobile agents can be created and then freely migrated as long as the security policy allows. The specification and enforcement of security policies defines a sandbox in which application code, agents in particular, execute. The specification of security policies is done through a language called SPL (Security Policy Language) [26]. SPL is then automatically translated into code that enforces that policy.

The most important OBIWAN data structures are illustrated in Fig. 1:

1. **Proxy-out/proxy-in pairs** [31]. 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. In Section 2.1, we describe how these proxies help in supporting object replication.
2. **Interfaces**. The interfaces implemented by each object and proxy-out/proxy-in pairs are presented now. **IA**, **IB**, and **IC**: these are the interfaces of objects A, B, and C, respectively, designed by the programmer; they define the methods that can be remotely invoked on these objects. **IProvider**: interface with methods `get` and `put` that supports the creation and update of replicas; method get results in the creation of a replica and method put is invoked to send a replica back to the process where it came from (in order to update its master replica). **IDemander** and **IDemandee**: interfaces that support the incremental replication of an object graph (as



```
interface IProviderRemote
  Object get(mode, depth)
  void put(Object)

interface IDemandee
    void setProvider( IProvider);
    void setDemander( IDemander);
    Object demand();

interface IProvider
  Object get( mode, depth)
  void put(Object)

interface IDemander
void setProvider( IProvider );
void updateMember( Object replica,
                   Object member );
```

Fig. 1. OBIWAN data structures.

described in Section 2.1.1). **IProviderRemote**: remote interface that inherits from IProvider so that its methods can be invoked remotely.

3. **Agent's home/mobile proxies**. An agent's home proxy offers, among other facilities, agent location independence to the application. So, to interact with agents, an application is not obliged to know where they reside. It only needs to interact with the corresponding home proxy, which will then forward the requested operations to the appropriate agent. So, an agent's home proxy is similar to a stub used for RMI.

   Cooperating with an agent's home proxy, there is a mobile proxy residing in the same host where the agent is being executed. The mobile proxy can be viewed as an extension of the home proxy in the remote host. As opposed to the home proxy, the remote one is mobile and travels between hosts, permanently accompanying the corresponding agent.

4. **GC-stubs and GC-scions**. A GC-stub describes an outgoing interprocess reference, from a source process to a target process (e.g., from object A in P2 to object Z in P3). A GC-scion describes an incoming interprocess reference, from a source process to a target process (e.g., to object Z in P3 from object A in P2).

   It is important to note that GC-stubs and GC-scions do not impose any indirection on the native reference mechanism. In other words, 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 stubs and scions (or skeletons) used for RMI (also represented in Fig. 1) that are managed by the underlying virtual machine.

5. **inPropList and outPropList**. These lists indicate the process *from which* each object has been replicated, and the processes *to which* each object has been

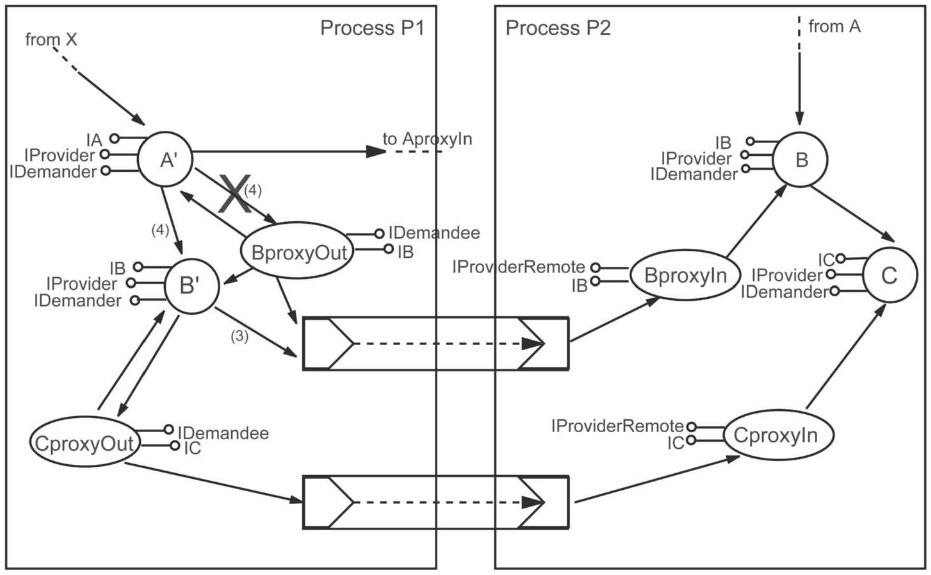


Fig. 2. Replication of B from P2 to P1 (numbers corresponding to enumerated items in Section 2.1.1).

replicated, respectively. Thus, 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, and **sentUmess**/**recUmess** is a bit indicating if a unreachable message (for DGC purposes) has been sent/received (more details are given in Section 2.2.1).

6. **Security module**. This module includes the event interceptor through which all relevant events are filtered (e.g., object invocation, agents' migration, etc.). Those events that are important for the enforcement of history-based policies are logged in the event log module. Events that require an authorization, before being effectively performed, are directed to the security monitor. The input for this last module is the specification of the policy being enforced and some context information that may be relevant to determine if the event should be authorized or not.

2.1 Replication

The application programmer, if he wants to, can control, both at compile-time and at runtime, which objects should be invoked remotely or locally. So, at any time, both replicas, the master and the local, can be freely invoked; the programmer decides what the best option is.

A local replica A′ can be updated from its master A, or update it, whenever the programmer wants. Obviously, due to replication, the issue of replicas' consistency arises. We leave the responsibility of maintaining (or not) the consistency of replicas to the programmer.[3]

The incremental replication of an object graph has two clear advantages with regard to the replication of the whole reachability graph in one step: 1) the latency imposed on the application is smaller because the application can invoke immediately the new replica without waiting for the whole graph to be available, and 2) only those objects that are really needed become replicated.

Thus, 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 available, are those in which incremental replication is useful. On the other hand, 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 and the network connection will not be available in the future, it is better to replicate the transitive closure of the graph. The application can easily make this decision at runtime, between incremental or transitive closure replication mode, by means of the mode argument of the method IProvideRemote::get(mode,depth).[4]

2.1.1 Incremental Replication

Without loss of generality, we describe how OBIWAN supports replication, taking into account the scenario illustrated in Figs. 1 and 2. There are two processes, P1 and P2, in two different sites, and the initial situation is the following:

1. P2 holds a graph of objects A, B, and C,
2. object A has been replicated from P2 to P1, thus we have A′ in P1,
3. A′ holds a reference to AproxyIn (for reasons that will be made clear afterward), and
4. given that B has not been replicated yet, A′ points to BproxyOut instead.

Note that A′ was replicated the same way that B will be, as explained afterward.

The stub-scion pairs for RMI support are created by the underlying virtual machine. Objects A, B, and C are created

3. Note that the application programmer is not forced to deal with consistency; he may simply use a library of specific consistency protocols [12], [17], [18], [20]. We plan to integrate such libraries in OBIWAN.

4. The runtime parameter "depth" specifies the depth of the graph to be replicated.

by the programmer; their replicas (A′, B′, etc.) are created either upon the programmer's request or automatically (i.e., as a result from object invocation). Proxies-in and proxies-out, as well as references pointing to them, are part of the OBIWAN platform and are transparent to the programmer.

Starting with the initial situation, the code in A′ may invoke any method that is part of the interface IB, exported by B, on BProxyOut (that A′ sees as being B′). For transparency, this requires the system to support a kind of "object faulting" mechanism as described now. All IB methods in BProxyOut simply invoke its demand method BProxyOut.demand (interface IDemandee) that runs as follows:

1. It invokes method BProxyIn.get in P2 (BProxyIn is BProxyOut's provider).
2. BProxyIn.get invokes B.get (interface IProvider) that will proceed as follows: It creates B′, CProxyOut, CProxyIn and sets the references between them; once this method terminates, B′, BProxyOut, and CProxyOut are all in P1, CProxyIn is in P2; note that A′ and BProxyOut still point to each other (Fig. 2 illustrates this situation and the following two, by enumerating the corresponding arrows).
3. 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).
4. BProxyOut invokes A′.updateMember(B′,this) so that A′ replaces its reference to BProxyOut with a reference to B′.
5. 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.
6. From this moment on, BProxyOut is no longer reachable in P1 and will be reclaimed by the garbage collector of the underlying virtual machine.

It is important to note that, once B gets replicated in P1, as described above, 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; this fault will be solved with a set of steps similar to those previously described. In addition, note that this mechanism does not imply the modification of the underlying virtual machine. This fact is key for OBIWAN portability.

The replication mechanism just described is very flexible in the sense that it allows each object to be individually replicated. However, this has a cost that results from the creation and transfer of the associated data structures (i.e., proxies). To minimize this cost, OBIWAN allows an application to replicate a set of objects as a whole, i.e., a cluster, for which there is only one proxy-out/proxy-in pair.

A cluster is a set of objects that are part of a reachability graph. For example, if an application holds a list of 1,000 objects, it is possible to replicate a part of the list so that only 100 objects are replicated and a single proxy-out/proxy-in pair is effectively created. Thus, the amount of objects in the cluster can be determined at runtime by the application. The application specifies the depth of the partial reachability graph that it wants to replicate as a whole. So, these clusters are highly dynamic. This is an intermediate solution between: 1) having the possibility of incrementally replicating each object or 2) replicating the whole graph. (See Section 4 for performance results of both approaches.)

2.2 Distributed Garbage Collection

Consider a scenario in which the initial situation is illustrated in Fig. 1. Now, suppose that, due to application execution in P1, A′ becomes locally unreachable[5] and, due to application execution in P2, A no longer points to Z. Then, the question is: Should Z be considered unreachable, i.e., garbage? As a matter of fact, Z must be considered to be reachable because it is possible for an application in P2 to update A from process P1 (recall that outPropList in P2 stores all the processes holding replicas of A). Thus, the fact that A′ is *locally* unreachable in process P1, and A no longer points to Z, does not mean that Z is *globally* unreachable. Therefore, a target object Z is considered unreachable only if the union of all the replicas of the source object, A in this example, do not refer to it. We call this the Union Rule (more details are given in Section 2.2.1).

Classical DGC algorithms (i.e., those designed for RMI-based systems) erroneously consider that Z is effectively garbage, i.e., that it can be deleted. Larchant [11] does handle this situation; however, it imposes severe constraints on scalability because it requires the underlying communication layer to support causal delivery [14]. In OBIWAN, we provide an algorithm for DGC that, while being correct in presence of replicas (as Larchant), is more scalable because it does not require causal delivery to be provided by the underlying communication layer.

2.2.1 Algorithm

The DGC algorithm is a hybrid of tracing and reference-listing [23], [24], [32]. Thus, each process has two components: a local tracing collector and a distributed collector. Each process does its local tracing independently from any other process. The local tracing can be done by any mark-and-sweep based collector. The distributed collectors, based on reference-listing, work together by exchanging asynchronous messages.

The local and distributed collectors depend on each other to perform their job in the following way. A local collector running inside a process traces the local object graph starting from that process's local root and set of GC-scions. A local tracing generates a new set of GC-stubs, i.e., for each outgoing interprocess reference, it creates a GC-stub in the new set of GC-stubs. From time to time, possibly after a local collection, the distributed collector sends a message called `newSetStubs`; this message contains the new set of GC-stubs that resulted from the local collection; this message is sent to the processes holding the GC-scions corresponding to the GC-stubs in the previous GC-stub set. In each of the receiving processes, the distributed collector matches the just received set of GC-stubs with its set of GC-scions;

5. Locally (un)reachability means (un)accessibility from the enclosing process's local root (i.e., stack and static variables) and GC-scions.

those GC-scions that no longer have the corresponding GC-stub, are deleted.

Once a local tracing is completed, every locally reachable object has been found (e.g., marked, if a mark-and-sweep algorithm is used); objects not yet found are locally unreachable; however, they can still be reachable from some other process holding a replica of, at least, one of such objects. To prevent the erroneous deletion of such objects, the local collector traces the objects graph from the lists inPropList and outPropList. Thus, the local and distributed collectors perform as follows:

1. When a locally reachable object (already discovered by the local collector) is found, the tracing along that reference path ends.
2. When an outgoing interprocess reference is found, the corresponding GC-stub is created in the new set of GC-stubs.
3. For an object that is reachable only from the inPropList, a message `unreachable` is sent to the process from where that object has been replicated; this sending event is registered by changing a `sentUmess` bit in the corresponding inPropList entry from 0 to 1. When a `unreachable` message reaches a process, this delivery event is registered by changing a `recUmess` bit in the corresponding outPropList entry from 0 to 1.
4. For an object that is reachable only from the outPropList, and the enclosing process has already received a `unreachable` message from all the processes to which that object has been previously replicated, a `reclaim` message is sent to all those processes and the corresponding entries in the outPropList are deleted; otherwise, nothing is done. When a process receives a `reclaim` message it deletes the corresponding entry in the inPropList.

As already mentioned, an object can be reclaimed only when all its replicas are no longer reachable. This is ensured by tracing the objects graph from the lists inPropList and outPropList; objects that are reachable only from these lists are not locally reachable; however, they cannot be reclaimed without ensuring their global unreachability, i.e., that none of their replicas are accessible. (This is the basis for the Union Rule.)

Concerning the interaction between applications and the DGC algorithm, we have the following: 1) immediately before a message containing a replica is sent, the references being *exported* (contained in the replicated object)[6] must be found in order to create the corresponding GC-scions, and 2) immediately before a message containing a replica is delivered, the outgoing interprocess references being *imported* must be found in order to create the corresponding local GC-stubs.[7]

It is worthy to note that the DGC algorithm does not require the underlying communication layer to support causal delivery (which is an improvement with regard to Larchant). This clearly contributes to its scalability and is ensured because the DGC algorithm creates the corresponding GC-scions and GC-stubs immediately before a replica is sent and delivered, respectively.

Thus, the DGC algorithm can be summarized by the following safety rules:

1. **Clean Before Send Replica**. Before sending a message containing a replica of an object X from a process P, X must be scanned for references and the corresponding GC-scions created in P.
2. **Clean Before Deliver Replica**. Before delivering a message containing a replica of an object X in a process P, X must be scanned for outgoing interprocess references and the corresponding GC-stubs created in P.
3. **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.

2.3 Mobile Agents Security

For the definition of security policies in OBIWAN, we developed SPL [26]. This language is sufficiently expressive to define a vast group of authorization and obligation policies, is very efficient, and is enforceable by a security monitor. This expressiveness and efficiency allows its use in a wide variety of systems, particularly, in OBIWAN. Although flexible enough to express numerous complex security models [30], such as role-based access control (RBAC), discretionary access control (DAC), or obligation-based policies, we focus on its ability to express history-based security policies.

The SPL language is based on four essential entities: objects, groups, rules, and policies. The rules establish constraints through the relations among objects and groups. Policies result from the composition of multiple rules and groups, and are the main building blocks of SPL. This language is therefore policy-oriented and constraint-based.

SPL objects are typified with an explicit interface, through which their properties can be obtained and modified. These objects may represent not only internal authorization model objects, but also external platform resident objects. Although there are some internal objects and groups, the vast majority are external, such as mobile agents, hosts, or system resources (e.g., files). Each external object has an associated type. That type is used to define its interface and subsequent properties. As an example, each mobile agent is an object of the type mobileAgent, which has specific mobility properties.

The type definition in SPL is similar to the interface definition in Java. There are six basic types that can be used: number, string, Boolean, rule, policy, and object. The definition of types object and mobileAgent is presented in Fig. 3. Besides the definition of types by inheritance, it is also possible to define types based on the composition of groups and previously existing types. Such an example may be the definition of an agent society type as an alias for a group of agents: `alias agent group agentSociety;`.

Groups, in this platform, are fundamental since they allow the association of entities with similar properties, leading to the ease of policy definition and to the increased expressiveness and scalability of the language. SPL supports two different kinds of groups: sets and categories. Sets are groups that result from insertions and removals of individual

6. When an object is replicated to a process, we say that its enclosed references are **exported** from the sending process to the receiving process; on the receiving process, i.e., the one receiving the replicated object, we say that the object references are **imported**.

7. Note that this may result in the creation of chains of GC-stub/GC-scion pairs, as it happens with SSP Chains [32].

```
type object {
  string name;
  user owner;
  user creator;
  string homeHost;
  number timeOfCreation; }
```

(a)

```
type mobileAgent extends object {
  boolean running;
  string group previousHosts; }
```

(b)

```
DestinyRule: ce.source.type =
  mobileAgent &
  ce.source = "John" &
  ce.target.host = "P1" &
  ce.operation = "migration"
  :: true
```

(c)

Fig. 3. SPL examples: (a) definition of type object, (b) definition of type mobileAgent, and (c) rule applicable to all migration requests to host P1 that are generated by the mobile agent John.

```
Policy AgentChineseWall {
host set InterestClass;
?AgentChineseWall:
  EXIST e IN PastEvents {
    ce.target IN InterestClass & e.target IN InterestClass &
    ce.operation = "Migration" & e.operation = "Migration" &
    ce.target != e.target
    :: false }; }
```

(a)

```
Policy HistoryBasedServiceControl {
 ?HistoryBasedServiceControl:
   NOT EXIST e IN PastEvents {
     ce.source = "Bill" & ce.operation = "Invocation" &
     ce.operation.method = M2
     ::
     e.source = "Bill" & e.operation = "Invocation" &
     e.operation.method = M1 }; }
```

(b)

Fig. 4. Specification of history-based policies with SPL: (a) Chinese-wall and (b) interaction between two agents.

elements. Categories are groups defined by their elements' properties. For example, a category expressed in terms of the agents that have process P1 as home host is expressed as follows: `mobileAgent group homeAgents = AllAgents @ {.homeHost = P1};`.

Another important SPL entity is the rule. Rules are entities that establish constraints on the authorized operations. An authorization policy may therefore be expressed in terms of a set of rules. SPL rules are three-value logical expressions. They may assume the following values: allow, deny, and not apply. These values decide the acceptability of the events that are generated by the OBIWAN platform.

There are two classes of special events that deserve to be mentioned: 1) the current event and 2) past events. The first one is the event that is being checked and over which approval is requested. The second type of events are already approved or refused events that constitute the knowledge base for history-based security policies.

A rule is composed by two logical expressions. The first one defines the applicability domain of the rule, while the second expression sets the acceptability domain. For example, the rule in Fig. 3c is applicable to all migration requests to host P1 that are generated by the mobile agent John. The acceptability domain is always true, so the event is always allowed if the rule is applicable.

A policy is a set of rules and groups that determines the authorization and prohibition of a given domain of events. In each policy, there is a special rule, designated Query Rule (QR), identified by a question mark before the name of the rule, which is responsible for the policy's behavior and decisions. This rule may, in turn, call other rules in order to enforce the desired security policy.

2.3.1 Agents' History-Based Policies

OBIWAN supports the definition and enforcement of security policies on mobile agents and hosts. As our platform focuses on common and frequent security threats for mobile agent systems, such as denial-of-service by means of illegal resource usage, or undesired information flows, we need to offer protection mechanisms to both agents and hosts, in order to prevent that sort of attacks.

Thus, we support not only 1) mobile agent policies, which protect agents from potentially harmful operations, either due to careless agent programming, or due to interactions with nonreliable entities, such as hosts or other agents, but also 2) host policies that protect processes from undesired operations perpetrated by mobile agents.

Consider the situation where it is necessary to implement a Chinese-wall policy [3].[8] For this situation, suppose we have a single class of interest that contains multiple hosts. In this scenario, any agent that has already been executed in a given process, will be denied access to any other process in that class of interest. In Fig. 4a, we show the specification of this policy for OBIWAN.

Mobile agents must also be protected from interactions with other agents. Consider the simple scenario where mobile agent Joe offers two services, provided by methods M1 and M2. Joe is very cautious and, therefore, does not allow some agents, such as agent Bill, to access the service provided by M2 if it has already accessed service M1. This policy could be specified in OBIWAN as shown in Fig. 4b.

As stated before, OBIWAN policies may be defined not only for mobile agents, but also for hosts. Actually, these policies may be used to control the acceptability of agent's requested operations in those hosts. In particular, these policies may accept or refuse the arrival of a specific mobile agent to the host that enforces such policies. They can, additionally, control agent's access to host's protected system resources, such as files or network connections.

As an example, consider a history-based duty cooperation policy in which two hosts cooperate in some task with basic and advanced operations; the latter can only be performed once the basic operation has been performed. For that matter, suppose we have two different hosts, P1 and P2, that cooperate. In host P1, the mobile agent performs the basic operation of that task. After that, the mobile agent must migrate to host P2 in order to execute the advanced operation of the same task. However, the agent

8. In a Chinese-wall policy, every resource belonging to one class of interest can only be accessed by a user that has not previously accessed another resource of that same class of interest.

```
Policy HistoryCooperationDuty {
?HistoryCooperationDuty:
  EXIST e IN P1.PastEvents {
    ce.operation = "Migration"
    ::
    e.operation = "Basic" &
    e.source = ce.source &
    e.target = ce.target }; }
```

(a)

```
Policy DoS(number MaxOpenWin) {
 ?DoS: EXIST AT_LEAST MaxOpenWin e IN PastEvents {
       ce.source = e.source &
       ce.operation = "OpenWindow"
       :: deny; }; }
```

(b)

Fig. 5. Specification of history-based policies with SPL: (a) duty cooperation policy and (b) preventing a denial of service attack (abusive windows creation).

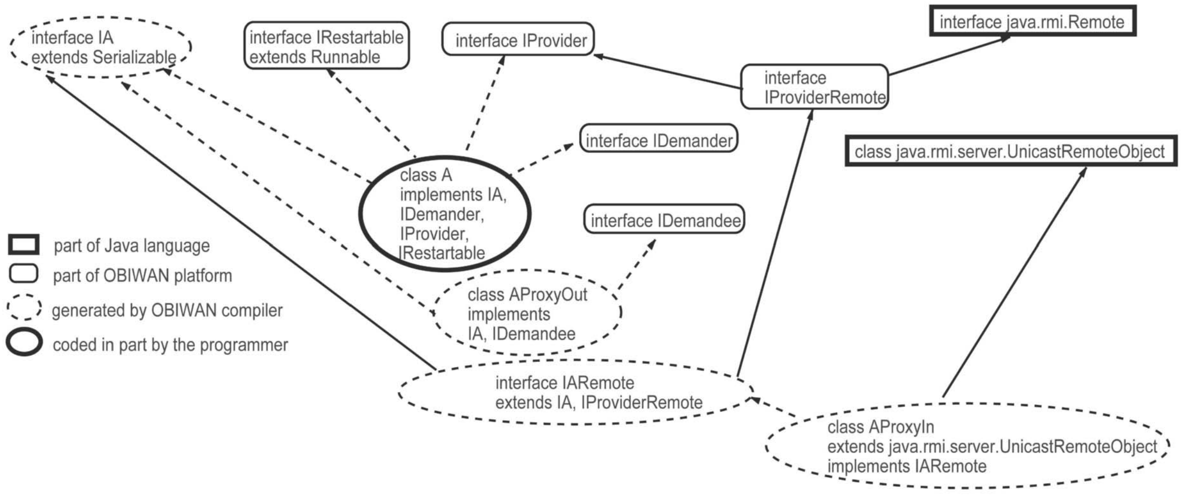


Fig. 6. Interfaces and classes in Java. Inheritance is represented with a solid line; implementation is represented with a dashed line.

should not migrate to host P2 before having executed the basic operation.

This policy is specified as shown in Fig. 5a. In this policy, host P2 searches host P1 past events for any event that corresponds to the basic operation performed by this agent, which is expressed by the constraint `e.source = ce.source`. Note that the constraint `e.target = ce.target` guarantees that the basic and the advanced operations refer to the same task.

Finally, to illustrate a denial of service attack, consider the policy definition in Fig. 5b; it prevents an agent to consume, without limitation, the resources associated with the creation of windows in the host process. As a matter of fact, this policy defines that an agent cannot perform an "OpenWindow" operation if the number of windows already created is equal to "MaxOpenWin."

3 Implementation

We developed two prototypes of OBIWAN: 1) one runs on top of the JVM and is written in Java, and 2) the other runs on top of the .Net CLR and is written in C#.[9] The differences between the two prototypes are minimal. Both JVM and the CLR support the basic functionality required, i.e., RMI, dynamic code loading and reflection. OBIWAN does not require any modification of either JVM or CLR internals. This fact is key for OBIWAN portability.

3.1 Classes and Interfaces

The most relevant interfaces and classes concerning replication are illustrated in Fig. 6 (recall Fig. 1 for the corresponding methods). The differences between Java and C# are minimal. Thus, hereafter, we will consider only the Java interfaces.

The "rectangular" interface and class are part of the regular Java distribution. The "rounded rectangles" represent OBIWAN platform interfaces that are constant and, therefore, precompiled. The "dashed ellipses" represent classes and interfaces automatically generated by **obicomp**.[10] Finally, the solid "ellipse" represents the class that the programmer writes.

The implementation of interfaces IDemander, IProvider and, if desired, IRestartable, is automatic through source code augmentation of class A. The programmer only has to write class A (note that the corresponding interface IA can be derived from it) and, obviously, the code of the client that invokes an instance of A. The interfaces IProvider and IProviderRemote are constant, thus they do not have to be generated each time an application is written. The interface IARemote and classes AProxyOut and AProxyIn are generated automatically.

OBIWAN provides support for the migration of execution flow through the interface IRestartable that is automatically implemented by obicomp. Programmers

9. Except the part that, from the security specification, generates the corresponding security monitor; this code is only written in Java, but can generate either Java or C#.

10. Obicomp is the OBIWAN tool that generates the code needed for replication, DGC, and security (from the policy specification).

just need to implement the `run` method of the `java.lang.Runnable` interface.

Since threads' stacks are not first class objects (both in .Net and Java), the programmer must provide synchronization points in which the agent execution can be frozen, its state serialized and transferred for ulterior reactivation upon arrival on another process. Thus, at certain points of execution, the programmer must invoke the checkpoint method of the IRestartable interface (recall that all methods of this interface are automatically implemented). The checkpoint method implements a synchronization point where it is safe to freeze the execution flow on an object, serialize its data, transmit it, and reactivate 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.

To summarize, when a new application is developed the programmer does the following steps: 1) write the interface IA, 2) write the class A, and 3) run obicomp. The last step automatically generates the other interfaces and classes needed, and extends class A implementing interfaces IProvider and IDemander. Additionally, the support for the migration of execution flow, i.e., agents, is achieved simply by having class A implement the interfaces Runnable (provided by Java) and IRestartable (provided by OBIWAN); OBIWAN automatically generates the code that implements IRestartable. (Obviously, the programmer has to write method run.)

Currently, obicomp uses a mix of: 1) reflection to analyze classes and generate the corresponding proxies, and 2) source code insertion to augment the classes written by the programmer with the methods that implement interfaces IDemander, IProvider, and IRestartable.

3.2 Distributed Garbage Collection

Basically, the code of the distributed garbage collector implements the safety rules (recall Section 2.2.1). The implementation of these rules consists mostly on scanning the objects being replicated and creating the corresponding GC-scions and GC-stubs.

An important aspect concerning the implementation of the distributed garbage collector is the data structures supporting the GC-stubs and GC-scions. These were conceived taking into account their use, in particular, to optimize the kind of information exchanged between processes that occurs when a message with a new set of GC-stubs is sent. This message carries the new set of GC-stubs, resulting from a local collection; it is sent to the processes holding the GC-scions corresponding to the GC-stubs in the previous GC-stub set. Then, in each of the receiving processes, the distributed collector matches the just received set of GC-stubs with its set of GC-scions; those GC-scions that no longer have the corresponding GC-stub are deleted.

Thus, GC-stubs are grouped by processes, i.e., there is one hash table for each process holding GC-scions corresponding to the GC-stubs in that table. Sending a new set of GC-stubs to a particular process is just a matter of sending the new hash table. The same reasoning applies to GC-scions: they are stored in hash tables, each table grouping the GC-scions whose corresponding GC-stubs are in the same process.

3.3 SPL

Given the resemblance of SPL and Java/C# structure, most of the compiler actions are simple translations: Each SPL policy is directly translated into a Java/C# class; each rule is translated into a trivalue function without parameters (with the exception of the query rule which has one parameter: the current event); each object is translated into a Java/C# interface; and each group variable is translated into a Java/C# variable of type "SplGroup," which defines an interface to access several kinds of groups (external groups, subgroups of external groups, and internal groups).

Wherever a policy instance is used in place of a rule, the obicomp executes an automatic cast operation consisting of making explicit the call to the query rule of the policy. Thus, the overall structure of the generated code can be seen as a tree of trivalue functions calling other functions, in which the root is the function resulting from the translation of the query rule of the master policy and the leaves are the functions resulting from simple rules.

4 Evaluation

We evaluated the OBIWAN middleware platform by developing new applications, porting existing ones, and measuring its performance. However, for lack of space, we only show some relevant performance results of OBIWAN concerning its core functionality: 1) the cost of incremental object replication with and without clustering, 2) the performance penalty due to DGC safety rules, and 3) the cost of evaluating a large number of SPL rules.

All the results were obtained in a 100 Mb/sec local area network, connecting several PCs with Pentium II and Pentium III processors, either with 64 Mb or 128 Mb of main memory each, running JDK 1.3 on top of Windows 2000.

4.1 Incremental Replication

We present the performance of incremental replication for objects with 64 bytes and 1,024 bytes (more results in [37]). We use a list with 1,000 objects (all with the same size) that is created in process P2. This list is then replicated into another process P1, in several steps, each step replicating 1, 25, 100, 250, 500, or 750 objects. Then, the application running in P1 invokes a dummy method on each object of the list. When the object being invoked is not yet replicated the system automatically replicates the next 1, 25, 100, 250, 500, or 750 objects.

The results are presented in Fig. 7a. Note that, the time values include the creation and transfer of the replicas along with the corresponding proxy-out/proxy-in pairs for each object being replicated. So, in this case, i.e., without clustering, each object still can be individually updated in P2.

From Fig. 7a, we can conclude that:

1. the steps observed are due to the creation and transfer of several replicas along with the corresponding proxy-out/proxy-in pairs,
2. the incremental replication of one object individually at each time is the most flexible alternative, but is the least efficient for large number of invocations,
3. the incremental replication of 25 to 100 objects at each time is the most efficient alternative,

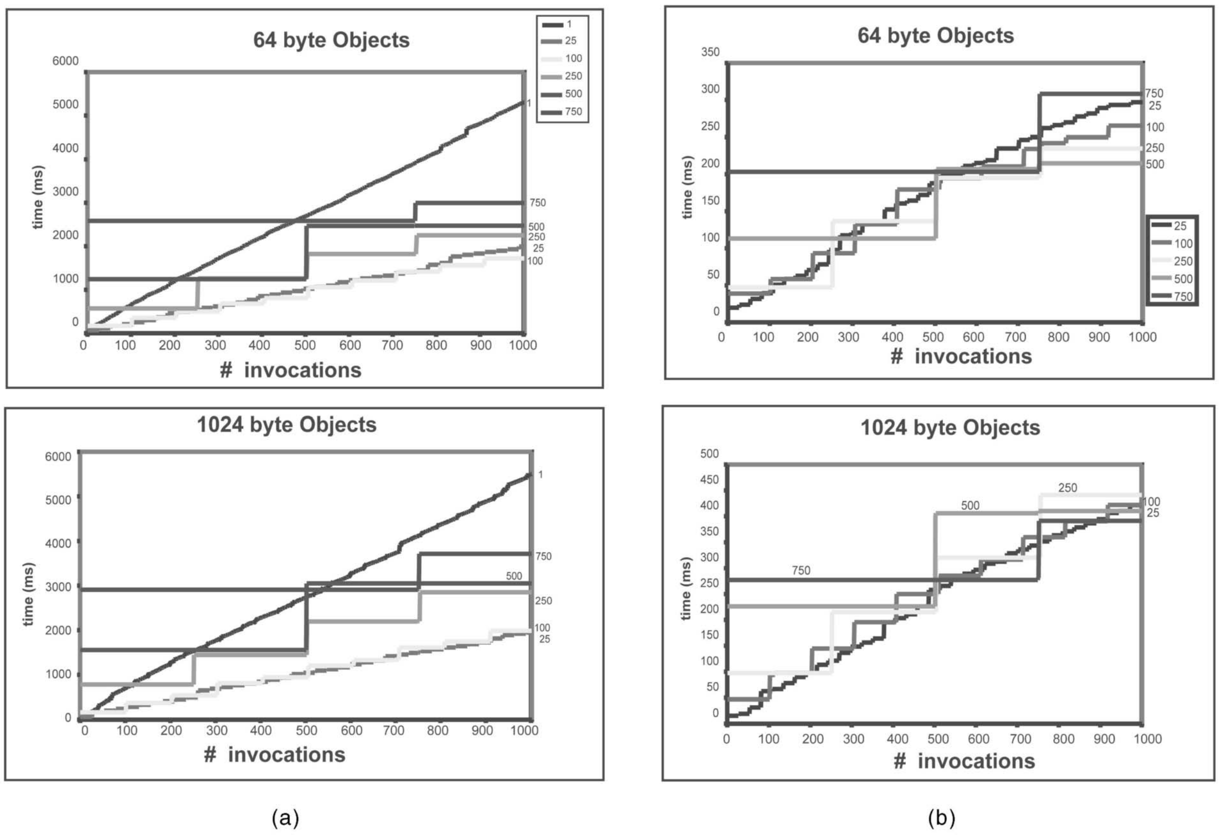


Fig. 7. Performance results for replication. (a) Incremental replication without clustering and (b) incremental replication with clustering.

4. the incremental replication of 500 or 750 objects at each time is not efficient because of the high cost of creation and transfer of the corresponding replicas and proxy-out/proxy-in pairs, and
5. for computers with a small amount of free memory, when only a part of the objects are effectively needed, it is clearly advantageous to incrementally replicate a small number of objects (but, more than one at each time).

To obtain the performance of incremental replication with object clustering, we used the same approach; the list and object sizes are those previously mentioned. The application running in process P1 invokes a method on each object of the list. 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 one of these clusters, all objects are replicated as a whole, thus there is only one proxy-out/proxy-in pair being created. Consequently, each object cannot be individually updated in P2.

The results are presented in Fig. 7b. Note that, in each case, the time values include the creation and transfer of all the replicas along with the single corresponding proxy-out/proxy-in pair.

We can conclude that, with regard to the results without clustering, these results are: 1) much better because there is only one proxy-out/proxy-in pair being created for each cluster; in addition, we observed that the most significant performance cost remaining in the cluster replication mechanism is due to data serialization (done by the JVM) and network communication,[11] and 2) not that sensitive to the amount of objects being replicated at each time (i.e., the curves are closer); the reason is the same as in 1).

4.2 Distributed Garbage Collection

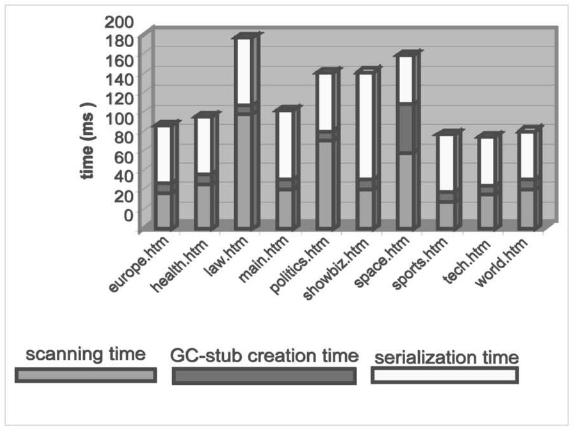
We exercised the OBIWAN platform with several applications. In one of them, News Gathering (NG), Web pages are treated as objects (instances of classes), i.e., a given Web page written in HTML can be freely replicated. (More details of NG and DGC are in [27], [38].) When compared to applications in which an object is an instance of a Java/C# class, the relevant difference is that references are, in fact, URLs.

The critical performance results are those related to the implementation of safety rules 1 and 2. Thus, we downloaded a part of the graph of objects of a well-known Web site (cnn.com) and, for each one, ran the code implementing the safety rules;[12] more precisely, we downloaded 155 HTML files and obtained for each one the time it takes to: scan it, create the corresponding GC-stubs, and serialize the hash table containing the GC-stubs (including writing to disk). For clarity, we only present the time it takes to create GC-stubs and their size because the same values apply to GC-scions.

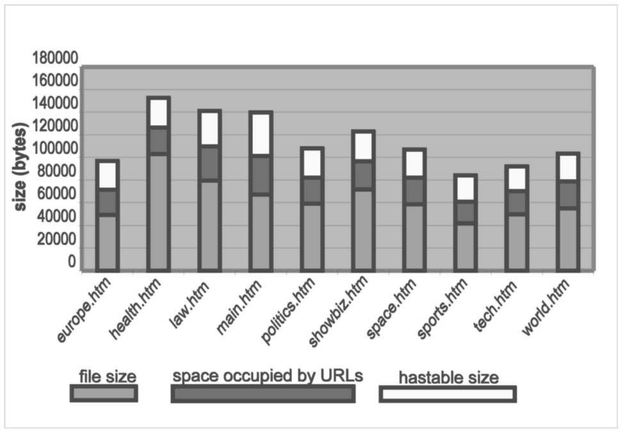
For the 155 files, we obtained the following results: the mean file size (43,563 bytes), the mean number of URLs

11. For example, with clustering, the cost of creating 500 replicas in P2, 1,024 bytes each, is about 10 ms.

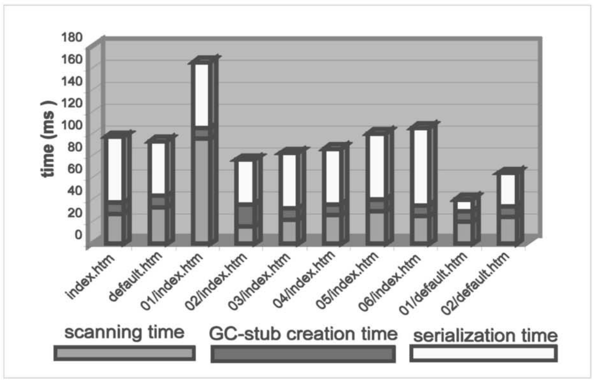
12. We used a depth of 5 (equivalent to the second argument of method IProviderRemote::get(mode,depth)) because it provides a large number of files without getting all the site.



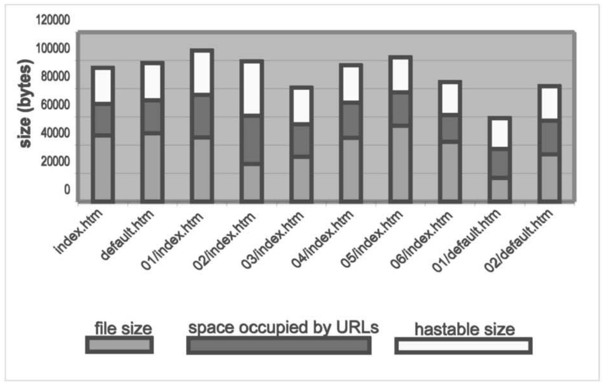
(a)



(c)



(b)



(d)

Fig. 8. Performance results for DGC. (a) Time spent for the top group, (b) time spent for the branch group, (c) space occupied for the top group, and (d) space occupied for the branch group.

enclosed in each file (326), the mean time to scan a file (38 ms), the mean time it takes to create a GC-stub in the corresponding hash table (3 ms), the mean size of the hash table containing all the GC-stubs corresponding to all the URLs enclosed in a file (19,252 bytes),[13] and the mean time it takes to serialize a hash table with all the GC-stubs corresponding to a single file (67 ms).

However, in a real situation, we expect that only some objects get replicated. A possible user would access a few top-level pages and then pick one or more branches of the hierarchy and follow them down. Some of these files would be replicated into the user's computer. So, in order to obtain more realistic numbers, we performed the following: We picked 10 files from the top of the cnn.com hierarchy. These files are mostly entry points to the others with more specific contents. We call this set of files the **top-set**. We also picked other 10 files representing a branch of the cnn.com hierarchy, world/Europe. We call this set of files the **branch-set**. In Figs. 8a and 8b, we present, for each file of the two considered sets, the time spent in each relevant operation. In Figs. 8c and 8d, we present, for the same files in the same two sets, the space occupied by: the files themselves, the URLs enclosed in them, and the hash table containing the corresponding GC-stubs.

13. This value depends mostly of the size of the corresponding URLs.

These performance results are worst-case because they assume all the URLs enclosed in a file refer to a file in another site, which is not the usual case. However, they give us a good notion of the performance limits of the current implementation. In particular, we see that the most relevant performance costs are due to file scanning and hash table serialization. However, we believe that these values are acceptable, taking into account the functionality of the system, i.e., it ensures that no distributed broken links or memory leaks occur. In addition, when a user runs the NG browser and accesses any Web page without making a local replica of any file, there is no performance overhead due to DGC.

4.3 Security

One of the problems of expressive security frameworks such as ours is the low efficiency of their implementations. While, in common access control list (ACL) based systems [13], only the access control entries (ACE) belonging to the ACL of each target object are evaluated on each access, in OBIWAN, potentially every rule has to be evaluated for every access. This is a problem in systems with thousands of rules, users, and objects. However, in common security policies, for each event, only a small number of rules is applicable (i.e., the applicability domain is false for that event); thus, most of the rules are not applicable and do not need to be evaluated.

In OBIWAN, we have designed a simple target-based index for rules, which is able to decide, based on the target of each event, which rules are applicable and which are not. This index technique has proven to be efficient. In particular, for a policy with 4,120 rules, 12,000 targets, and 5,000 users, the time to evaluate the rules, for authorization purposes of an event, is $40\mu s$, which is about 6 percent of the time to open a file (for reading).

Although the index technique proved its efficiency, even to large security policies, it is not enough to ensure efficiency of history-based policies. History-based policies need to record and query recorded information about past activities. Some security services record events implicitly in their own data structures [29] (mostly using labels), others record them explicitly into an event log [2]. The former solution lacks the flexibility necessary to the SPL history-based expressiveness, and the latter suffers from the common log-size problem.

In OBIWAN, we solved this problem by having one log for each history-based rule instead of having one single global log for every rule. Therefore, each log is specially tuned for the rule it serves, by keeping only the information required by that rule; this information is kept inside a log structure specifically crafted for the type of search required. In particular, repeated events are accounted by simply incrementing the corresponding counter. Thus, most history-based policies have bounded logs because only a few events needs to be logged. However, currently, history-based policies comparing event's instance times may still be unbounded.

These logs are automatically created by obicomp when generating the code for the specified policy; thus, they are completely transparent to the programmer (or to the policy administrator). This solution proved to be efficient for several history-based policies. Namely, for a Chinese-wall policy with 10 classes of interest, the time to evaluate the policy, for authorization purposes of an event, is $43\mu s$ (in the worst possible scenario, where every access is allowed); this value is independent of the number of events recorded, and the log entry number is bound to the number of users accessing targets in the classes of interest.

5 Related Work

The OBIWAN platform can be related to several other systems that support remote invocation, replication, DGC, mobile agents, and security. An important difference is that such systems do not provide an integrated platform supporting all the mechanisms as OBIWAN does: paradigm flexibility (RMI, replication, mobile agents), automatic replication, DGC (correct in the presence of replicas), and security policies. This integration is an advantage to the programmer as he may decide what functionality is best adapted to his application scenario.

Javanaise [5], [15] is a platform that aims at providing support for cooperative distributed applications on the Internet. In this system, the application programmer develops his application as if it were for a centralized environment, i.e., with no concern about distribution. Then, the programmer configures the application to a distributed setting; this may imply minor source code modifications. A proxy generator is then used to generate indirection objects and a few system classes supporting a consistency protocol. Javanaise does not provide support for incremental replication; clusters are defined by the programmer and are less dynamic than in OBIWAN. In addition, Javanaise provides no support for security policy definition, mobile agents, or DGC.

There has been some effort in the context of CORBA to provide support for replicated objects [9] as well as in the context of the World Wide Web [5]. However, most of this work addresses other specific issues such as group communication, replication for fault-tolerance, protocols evolution, etc.

Thor [21] is a distributed object oriented database (OODB) that provides a hybrid and adaptive caching mechanism handling both pages and objects; it provides its own programming language and DGC (which does not consider object replicas).

Most OODBs [39], e.g., O_2 [6] and GemStone [4] are very heavyweight, and often come with their own specialized programming language. In addition, these systems offer the programmer a single programming paradigm and do not consider the security aspects.

There are a great number of Java-based mobile-agent systems available [19]. Among these, Object Space's Voyager [35] is the most interesting; it was designed from the ground to support object mobility. A Voyager's agent is simply a special kind of object that has the ability to move; otherwise, it behaves exactly like any other object. Voyager has introduced the concept of Virtual Object, which represents a proxy of a remote object or agent. Voyager can transform into an agent any arbitrary object using the Virtual Code Compiler. Once the object is processed, it exhibits some properties of an agent: it can be migrated from host to host and accessed remotely by other virtual objects in RMI-like fashion, and it can have its own life cycle. Unless specifically designed to be otherwise, they are simply passive objects that can be moved and manipulated remotely. In conclusion, with such agent platforms, the programmer can develop his application either with mobile agents or RMI. However, these platforms support neither objects to be automatically replicated nor provide DGC. In addition, the security specification and enforcement simply relies on native mechanisms (JVM or operating system).

Previous work in DGC such as IRC [23], SSP chains [32], and Larchant [11] served as the starting point of the DGC algorithm presented in this paper. Our algorithm is an improvement over these in such a way that it combines their advantages: no need for causal delivery support to be provided by the underlying communicating layer (from the first two), and capability to deal with replicated objects (from Larchant).

A work on DGC also related to ours is Skubiszewski and Valduriez GC-consistent cuts [34]. They consider asynchronous tracing of an OODB, but with no distribution or replication support. The collector is allowed to trace an arbitrary database page at any time, subject to the following ordering rule: For every transaction accessing a page traced by the collector, if the transaction copies a pointer from one page to another, the collector either traces the source page before the write, or traces both the source and the destination page after the write. In a certain way, these operations are equivalent to our safety rules 1 and 2. The authors prove that this is a sufficient condition for safety and liveness.

Concerning agents security, Deeds [8] offers an history-based access control mechanism that protects hosts resources from mobile code. In this platform, access control policies are written in Java. These policies can be inserted, removed, or simply modified while the monitored programs are still executing. An important limitation of Deeds is the way it identifies a program. It concatenates all user-level code for a mobile program and uses a hash algorithm to generate a name for it. This method has a major drawback: It does not allow the execution of programs that dynamically load new classes. This approach is not, therefore, appropriate for a mobile agent platform, since it does not support agent migration with code on demand.

Ponder [7] 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 RBAC policies. The obligation policies provided by Ponder are used in an agent platform [22] to specify mobility policies of agents. In this platform, application logic is completely separated from migration logic. Although designed for mobile agents, this platform still does not consider any type of support for history-based security policies.

All these security solutions have their own merit, but they all fail to provide an integrated middleware platform with the flexibility of OBIWAN. In addition, and specifically with regard to the implementation of the security modules, OBIWAN provides more functionality with good performance.

6 Conclusion

We presented OBIWAN, a middleware platform that helps programmers to develop distributed applications by allowing them to focus on the application logic. System-level issues such as object replication, preventing abusive resource consumption by mobile agents, and DGC, are automatically handled by the system. The performance results concerning replication, DGC, and security are very encouraging.

Programmers are free to use the programming paradigm that is most suited to their applications, either classical RMI, replication, or mobile agents. In particular, it is possible to change at runtime how objects are invoked: RMI or local invocation on a replica. Replicas are transparently created and mapped into processes; the programmer can control, at runtime, the amount of objects being replicated by creating dynamic clusters, thus improving the performance of the system.

OBIWAN provides a security framework (from the specification to the enforcement by means of a security monitor) that supports history-based policies. These can be applied to mobile agents and, for example, prevent the abusive resource consumption on hosts, or to enforce a Chinese-wall policy, among others.

In addition, OBIWAN supports distributed garbage collection that handles correctly multiple replicas of objects, thus releasing the programmer from an extremely error-prone task.

Concerning future work, we intend to address both practical and more theoretical issues. Thus, we plan to integrate the OBIWAN platform with software development tools (e.g., VisualStudio-C#, Eclipse-Java), to use a library of consistency and reconciliation of replicas, and we are currently integrating our reference-listing garbage collector within ROTOR (shared-source version of Microsoft .Net). Regarding the theoretical issues, we are working toward an extension of the distributed garbage collector in order to reclaim distributed cycles of garbage; we also intend to address the issue of fault-tolerance, and the specification of automatic policies adapted to the specific context environment (e.g., network latency and cost).

Acknowledgments

This work was partially funded by Microsoft Research.

References

[1] K. Arnold and J. Gosling, *The Java Programming Language.* Addison-Wesley, 1996.

[2] R. Simon and E. Zurko, "Adage: An Architecture for Distributed Authorization," OSF Research Inst., Cambridge, http://www.osf.org/www.adage/adage-arch-draft/adage-arch-draft.ps, 1997.

[3] D.F. Brewer and M.J. Nash, "The Chinese Wall Security Policy," *Proc. Symp. Research in Security and Privacy,* pp. 206-214, May 1989.

[4] P. Butterwoth, A. Otis, and J. Stein, "The GemStone Object Database Management System," *Comm. ACM,* vol. 34, no. 10, pp. 64-77, Oct. 1991.

[5] S.J. Caughey, D. Hagimont, and D.B. Ingham, "Deploying Distributed Objects on the Internet," *Recent Advances in Distributed Systems,* S. Krakowiak and S.K. Shrivastava, eds., Springer Verlag, Feb. 2000.

[6] O. Deux et al., "The O_2 System," *Comm. ACM,* vol. 34, no. 10, pp. 34-48, Oct. 1991.

[7] N. Dulay, E. Lupu, M. Sloman, and N. Damianou, "A Policy Deployment Model for the Ponder Language," *Proc. Seventh IEEE/IFIP Int'l Symp. Integrated Network Management,* 2001.

[8] G. Edjlali, A. Acharya, and V. Chaudhary, "History-Based Access Control for Mobile Code," *Proc. Fifth ACM Conf. Computer and Comm. Security,* pp. 38-48, Nov. 1998.

[9] P. Felber, R. Guerraoui, and A. Schiper, "Replication of CORBA Objects," *Recent Advances in Distributed Systems,* S. Krakowiak and S.K. Shrivastava, eds., Springer Verlag, Feb. 2000.

[10] P. Ferreira and M. Shapiro, "Garbage Collection and DSM Consistency," *Proc. First Symp. Operating Systems Design and Implementation,* pp. 229-241, 1994.

[11] P. Ferreira and M. Shapiro, "Modelling a Distributed Cached Store for Garbage Collection: The Algorithm and Its Correctness Proof," *Proc. Eighth European Conf. Object-Oriented Programming,* July 1998.

[12] K. Gharachorloo, S.V. Adve, A. Gupta, J.L. Hennessy, and M.D. Hill, "Programming for Different Memory Consistency Models," *J. Parallel and Distributed Computing,* vol. 15, no. 4, pp. 399-407, 1992.

[13] H.M. Gladney, "Access Control for Large Collections," *ACM Trans. Information Systems,* vol. 15, no. 2, pp. 154-194, Apr. 1997.

[14] R. Guerraoui and A. Schiper, "Total Order Multicast to Multiple Groups," *Proc. 17th Int'l Conf. Distributed Computing Systems,* pp. 578-585, 1997.

[15] D. Hagimont and F. Boyer, "A Configurable RMI Mechanism for Sharing Distributed Java Objects," *Internet Computing,* vol. 5, Jan. 2001.

[16] G. Karjoth, D.B. Lange, and M. Oshima, "A Security Model for Aglets," *Internet Computing,* vol. 1, no. 4, pp. 68-77, 1997.

[17] P. Keleher, A.L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," *Proc. 19th Int'l Symp. Computer Architecture,* pp. 13-21, May 1992.

[18] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel, "The Icecube Approach to the Reconciliation of Divergent Replicas," *Proc. 20th ACM Symp. Principles of Distributed Computing,* Aug. 2001.

[19] D.B. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[20] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems,* vol. 7, no. 4, pp. 321-359, Nov. 1989.
[21] B. Liskov, M. Day, and L. Shrira, "Distributed Object Management in Thor," *Proc. Int'l Workshop Distributed Object Management,* pp. 1-15, Aug. 1992.
[22] R. Montanari and G. Tonti, "A Policy-Based Infrastructure for the Dynamic Control of Agent Mobility," *Proc. IEEE Third Int'l Workshop Policies for Distributed Systems and Networks,* June 2002.
[23] J.M. Piquer, "Indirect Reference-Counting—A Distributed Garbage Collection Algorithm," *Proc. Conf. Parallel Architectures and Languages Europe,* pp. 150-165, June 1991.
[24] D. Plainfossé and M. Shapiro, "A Survey of Distributed Garbage Collection Techniques," *Proc. Int'l Workshop Memory Management,* Sept. 1995.
[25] D.S. Platt, *Introducing the Microsoft.NET Platform.* Microsoft Press, 2001.
[26] C. Ribeiro, A. Zúquete, P. Ferreira, and P. Guedes, "SPL: An Access Control Language for Security Policies with Complex Constraints," *Proc. Network and Distributed System Security Symp.,* Feb. 2001.
[27] A. Sanchez, L. Veiga, and P. Ferreira, "Distributed Garbage Collection for Wide Area Replicated Memory," *Proc. Sixth USENIX Conf. Object-Oriented Technologies and Systems,* Jan. 2001.
[28] R. Sandhu, "Separation of Duties in Computarized Information Systems," *Proc. IFIP WG11.3 Workshop Database Security,* Sept. 1990.
[29] R. Sandhu, "Lattice-Based Access Control Models," *Computer,* vol. 26, no. 11, Nov. 1993.
[30] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman, "Role-Based Access Control Models," *Computer,* vol. 29, no. 2, pp. 38-47, 1996.
[31] M. Shapiro, "Structure and Encapsulation in Distributed Systems: The Proxy Principle," *Proc. Sixth Int'l Conf. Distributed Systems,* pp. 198-204, May 1986.
[32] M. Shapiro, P. Dickman, and D. Plainfossé, "Robust Distributed References and Acyclic Garbage Collection," *Proc. 11th ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing,* 1992.
[33] R.T. Simon and M.E. Zurko, "Separation of Duty in Role-Based Environments," *Proc. IEEE Computer Security Foundations Workshop,* pp. 183-194, 1997.
[34] M. Skubiszewski and P. Valduriez, "Concurrent Garbage Collection in O2," *Proc. 23rd Int'l Conf. Very Large Databases,* M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, and M.A. Jeusfeld, eds. pp. 356-365, Morgan Kaufman, 1997.
[35] A. Silva, M. Mira da Silva, and J. Delgado, "An Overview of AgentSpace: A Next-Generation Mobile Agent System," *Proc. Second Int'l Workshop Mobile Agents,* Sept. 1998.
[36] A. Tripathi and N. Karnik, "Protected Resource Access for Mobile Agent-Based Distributed Computing," *Proc. ICPP Workshop Wireless Networking and Mobile Computing,* 1998.
[37] L. Veiga and P. Ferreira, "Incremental Replication for Mobility Support in OBIWAN," *Proc. 22nd Int'l Conf. Distributed Computing Systems,* pp. 249-256, July 2002.
[38] L. Veiga and P. Ferreira, "REPWEB: Replicated Web with Referential Integrity," *Proc. 18th ACM Symp. Applied Computing,* Mar. 2003.
[39] S. Zdonik and D. Maier, *Readings in Object-Oriented Database Systems.* San Mateo, Calif.: Morgan-Kaufman, 1990.

Paulo Ferreira received the PhD degree in computer science from the Université Pierre et Marie Curie (Paris-VI) in 1996. He received the MSc (1992) and BsEE (1988) degrees from the Technical University of Lisbon (IST/UTL, Instituto Superior Técnico), Portugal. He is a professor in the Computer and Information Systems Department at IST/UTL, where he has been teaching classes in the areas of distributed systems, operating systems, and Internet, both at the undergraduate and postgraduate levels. He has been a researcher at INESC since 1986, where he leads the Distributed Systems Group (www.gsd.inesc-id.pt). His research interests include system support for large-scale distributed data sharing, replication and consistency protocols, distributed garbage collection, persistence by reachability, security, operating systems, and Internet protocols. He is the author or coauthor of more than 40 peer-reviewed scientific communications and he has served on the program committees of several international conferences and workshops in the area of distributed systems. He is a member of the IEEE and IEEE Computer Society.

Luís Veiga received the BsCE degree in 1998 and the MSc degree in computer engineering in 2001, both from the Technical University of Lisbon (IST/UTL, Instituto Superior Técnico), Portugal. He is a PhD student and teaching assistant in the Computer and Information Systems Department at IST/UTL. He teaches operating systems and computer architecture classes. He has been a researcher at INESC-ID with the Distributed Systems Group since 1999. He has been an active participant in government and industry funded R&D projects such as Mnemosyne, MobileTrans, OBIWAN, and DGC-Rotor. His research interests include distributed systems, memory management for distributed and mobile computing, replication, distributed garbage collection, and mobility support. He has authored or coauthored 10 peer-reviewed scientific communications in workshops, conferences, and journals since 2000.

Carlos Ribeiro received the BsEE degree in 1989, the MSc degree in 1993, and the PhD degree in computer science in 2002, all from he Technical University of Lisbon (IST/UTL, Instituto Superior Técnico), Portugal. He is a professor in the Computer and Information Systems Department at IST/UTL, where he teaches operating systems and computer architecture classes. From 1995 to 1998, he was a security adviser for the Portuguese National Security Authority. He has been a researcher at INESC since 1988, where he has participated in several European projects. His main research area is security, although he is also interested in distributed operating systems and mobility.