Incremental Replication for Mobility Support in OBIWAN*

Luís Veiga Paulo Ferreira

INESC/IST, Rua Alves Redol 9, Lisboa, Portugal phone: 351 21 3100292
E-mail: {luis.veiga,paulo.ferreira}@inesc.pt

Abstract

The need for sharing is well known in a large number of distributed collaborative applications. These applications are difficult to develop for wide area (possibly mobile) networks because of slow and unreliable connections.

For this purpose, we developed a platform called OBIWAN[1] that: i) allows the application to decide, in run-time, the mechanism by which objects should be invoked, remote method invocation or invocation on a local replica, ii) allows incremental replication of large object graphs, iii) allows the creation of dynamic clusters of data, and iv) provides hooks for the application programmer to implement a set of application specific properties such as relaxed transactional support or updates dissemination.

These mechanisms allow an application to deal with situations that frequently occur in a (mobile) wide-area network, such as disconnections and slow links: i) as long as objects needed by an application (or by an agent) are co-located, there is no need to be connected to the network, and ii) it is possible to replace, in run-time, remote by local invocations on replicas, thus improving the performance and adaptability of applications.

The prototype is developed in Java, is very small and simple to use, the performance results are very encouraging, and existing applications can be easily modified to take advantage of OBIWAN.

1 Introduction

There is a clear need for data sharing and collaboration support in a large number of applications in different domains. In OBIWAN, we focus on applications in the area of co-operative work within virtual organizations; for example, a virtual enterprise grouping several companies from different countries, a virtual marketplace, a widely distributed software development team, a distributed game involving people anywhere in the world, etc.

This need for information sharing is increasing along two main axis: wide area (i.e., across the Internet) and mobility (i.e., portable computers, webpads, personal digital assistants, smart cellular phones, etc.). As a matter of fact, besides the growing number of desktop computers connected to the Internet, there are other devices, generally called information appliances (info-appliances, for short) that are gaining enormous popularity; personal digital assistants (PDAs) are just one of them.

The role of these info-appliances, currently handling agendas, calendars, etc. will certainly grow as more computing power and communications capability can be included [17, 19]. In particular, the foreseen increase of bandwidth in wireless communication makes the connection of these info-appliances to the Internet a reality [15].

We envisage a general scenario in which a user wants to access data using a 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 disconnections).

So, there is a constant need to access shared data no matter where you are and the info-appliance you use, and users want the same degree of responsiveness and performance as in a fully high-bandwidth low-latency wired connected environment. Sometimes these requirements may be impossible to fulfill but the system should be able to minimize the number of such occurrences.

For example, if accessing data on some remote machine is not possible for some reason, the application should not stop working; instead, it should, at least, automatically propose the user an alternative access to such data from another machine, even if such data is not up to date.

While disconnections maybe rare in stationary local area networks, they occur in greater number in mobile networks. Most applications consider them to be failures that are exposed to users. In the mobile environment, applications will face frequent, lengthy network disconnections. Some

*This work was supported by Microsoft Research.

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

Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)
1063-6927/02 $17.00 © 2002 **IEEE**

of these will be involuntary (e.g., due to a lack of network coverage) while others will be voluntary (e.g., due to a high dollar cost). Mobile applications should handle such disconnections gracefully and as transparently as possible. In addition, users should be able, as far as possible, to continue working as if the network was still available. In particular, users should be able to modify local replicas of global data.

Therefore, in this paper we focus on the following issues: incremental replication of an object graph, automatic object-fault detection and resolution.This paper is organized as follows. In the next section we present the architecture of OBIWAN focusing on incremental replication. In section 3 we describe the implementation of OBIWAN. In Sections 4 and 5 we present some experimental results and related work, respectively. Finally, in Section 6 we present some conclusions.

2 Architecture

OBIWAN gives to the application programmer the view of a network of machines in which one or more processes run; objects exist inside processes. An object can be invoked locally (after being replicated) or remotely (by means of RMI). OBIWAN is a set of runtime services on top of the JVM. The OBIWAN data structures are basically two: proxy-out/proxy-in pairs [18]. A proxy-out stands in for an object that is not yet locally replicated. For each proxy-out there is a corresponding proxy-in.

Without loss of generality, we now describe how OBIWAN works with a prototypical example shown in Figure 1: there are two processes in two different sites, S1 and S2, and the initial situation (a) is the following: i) S2 holds a graph of objects A, B and C; ii) only object AProxyIn is registered in a name server, and iii) S1 holds a remote reference to object AProxyIn, that was obtained from a name server. The prototypical example illustrates the most important data structures supporting the incremental replication of the objects graph from S2 to S1, and the corresponding object faults and their resolution.

Stubs and skeletons are created by the underlying virtual machine (Java in the current implementation). Objects A, B and C are created by the programmer; their replicas, A', B', and C' are created upon the programmer's request. All other objects, i.e. proxies-in and proxies-out, are part of the OBIWAN platform and are transparent to the programmer, except AProxyIn for reasons that will be made clear afterwards.

Figure 1 also shows, for each object and proxy, the interfaces implemented:

- IA, IB and IC: these are the remote interfaces of objects A, B and C, respectively, designed by the programmer; they define the methods that can be remotely invoked on these objects.

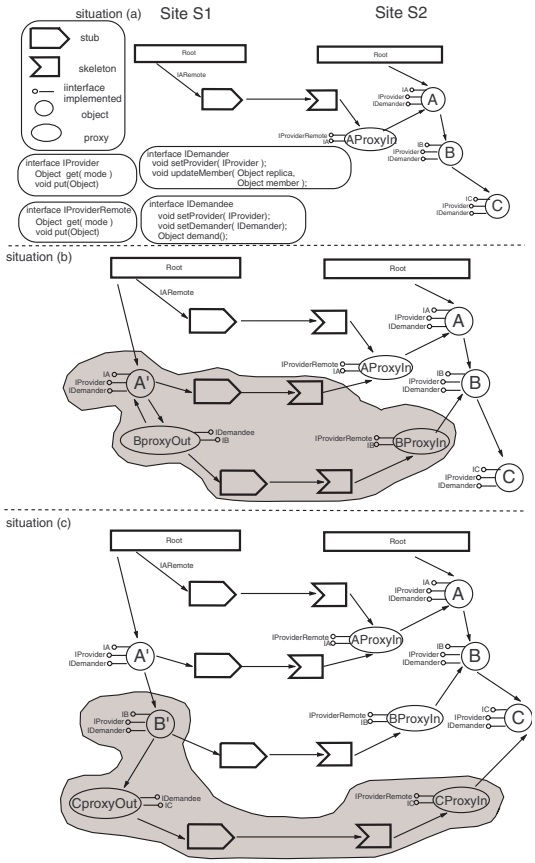

Figure 1. Replication in OBIWAN. Objects created on each step are shaded.

- 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 when a replica is sent 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's graph(v. subsection 2.2);
- IProviderRemote: remote interface that inherits from IProvider so that its methods can be invoked remotely.

2.1 Replication

Taking into account the OBIWAN architecture previously described, it is clear that OBIWAN allows the application programmer, if he wants so, to control, both at compile and at run-time, which objects should be invoked remotely or locally. An object that is invoked locally is a replica of a master in some remote process.

For example, in Figure 1, the master replica A, after being replicated into S1, can still be invoked via RMI because the local reference of type IARemote points to its master replica through AProxyIn. So, at any time, both replicas, the master and the local, can be freely invoked. It is the programmer, or even the user, who decide what the best option is.

A local replica A' can be updated from the 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.[2]

The incremental replication of an object graph has two clear advantages w.r.t. the replication of the whole reachability graph in one step: i) the latency imposed on the application is smaller because the application can invoke immediately the new replica,[3] and ii) only those objects that are really needed become replicated.

Thus, situations in which an application does not need to invoke all objects of a graph, or when the info-appliance where the application is running has limited memory 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, it is better to replicate the transitive closure of the graph. The application can easily make this decision in run-time, between incremental or "transitive closure" replication mode, by means of the mode argument of the method IProvideRemote::get(mode).

An architectural issue arises with the use of proxies-out. 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).

In the case of Figure 1, 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 S1 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 believe this is not 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 limitation also exists, for example, in Microsoft ActiveX components [3] and Java Beans [9].

[2]Note that the application programmer is not forced to deal with consistency; he may simply use a library of specific consistency protocols written by any other programmer. We plan to develop such libraries for well known consistency policies[13].

[3]Obviously, a perfect mechanism of pre-fetching in the background can completely eliminate the latency.

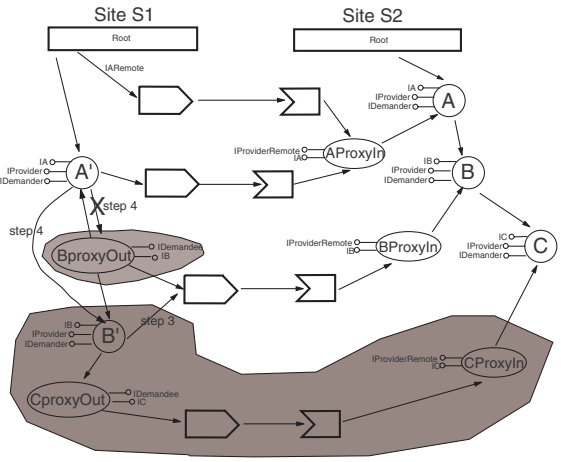

Figure 2. Intermediate step between situations (b) and (c) of Figure 1.

2.2 Incremental Replication

We now explain in detail the steps involved in the incremental replication of the graph presented in Figure 1. Starting with the initial situation (a), the application running in S1 requests A' by invoking method AProxyIn.get; this method simply invokes A.get. Then, this method executes the following:

1. create A' in S2;
2. for each reference A holds (only to B in this case) create the corresponding ProxyIn objects (only BProxyIn in this case) in S2; in the constructor of BProxyIn, set an internal reference pointing to B;
3. create a ProxyOut object for each ProxyIn created in the previous step (only BProxyOut in this case) in S2;
4. set the internal reference of A' (of type IB) so that it points to BProxyOut;
5. invoke BProxyOut.setProvider(BProxyIn) so that BProxyOut points to BProxyIn;
6. invoke BProxyOut.setDemander(A') so that BProxyOut also points to A'; return A'.

Thus, AProxyIn.get terminates simply by returning A'. As a result, A' and BProxyOut are automatically serialized by the underlying virtual machine and sent to S1. This results in situation (b) in Figure 1 where the just created objects are shaded. Later, 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 fault" mechanism as described now. All IB methods in BProxyOut simply invoke its demand method BProxyOut.demand that runs as follows(see Figure 2):

1. invokes method BProxyIn.get (BProxyIn is BProxyOut's provider);
2. BProxyIn.get invokes B.get that will proceed in a similar way as explained previously for A.get: creates B', CProxyOut, CProxyIn and sets the references between them; once this method terminates, as illustrated in Figure 2, B', BProxyOut and CProxyOut are all in S1, CProxyIn is in S2, and BProxyOut points to B'; note that A' and BProxyOut still point to each other;
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 S1 and will be reclaimed by the garbage collector of the underlying virtual machine.

It's important to note that, after situation (c) of Figure 1, 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 S1) an object fault occurs and will be solved with a set of steps similar to those previously described. The replication mechanism just described is very flexible in the sense that allows each object to be individually replicated. However, this has a cost that results from the creation and transference of the associated data structures (i.e., proxies). To minimize this cost OBIWAN allows an application to replicate a set of objects, i.e. a cluster.

A cluster is a set of objects that are part of a reachability graph. 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 sites. Thus, the amount of objects in the cluster is determined in run-time 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: i) having the possibility of incrementally replicate each object, or ii) replicating the whole graph. In Section 4 we present the performance results for all these possibilities.

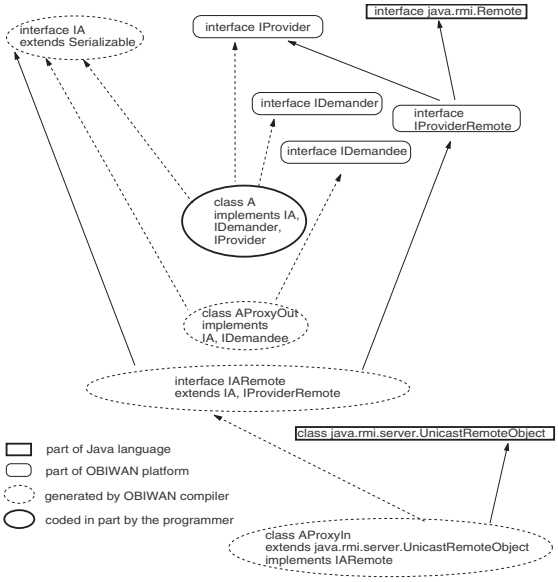

Figure 3. Interfaces and classes of OBIWAN. Inheritance is represented with a solid line; implementation is represented with a dashed line.

3 Implementation

The OBIWAN system runs on top of the Java virtual machine. Java is portable, free, simple to use, and supports the basic functionality required, i.e. RMI, dynamic code loading and reflection. In this section we describe the classes, interfaces and tools that constitute OBIWAN. We also illustrate how to port both legacy non-distributed and distributed-RMI applications on top of OBIWAN.

3.1 Classes and Interfaces

Suppose we want to build a distributed application with an object that can be either locally replicated or invoked via RMI. Additionally, we wanted objects to be replicated incrementally, i.e., as and only when they are needed.

All the interfaces and classes involved in this example are illustrated in Figure 3. The "rectangular" interfaces and classes are part of the regular Java distribution. The "rounded rectangles" represent OBIWAN platform interfaces that are constant and therefore pre-compiled. The "dashed ellipses" represent classes and interfaces automatically generated by the `obicomp` compiler. Finally, the solid "ellipse" represents the class that the programmer must write. The programmer only has to worry with the so-called "business-logic".

The implementation of interfaces IDemander and IProvider is automatic through source code augmentation

of the class the programmer has written. The programmer only has to write class A, the corresponding interface IA can be derived from it, and, obviously, the code of the client that invokes an instance of A. Note that 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.

To summarize, when a new application is developed the programmer does the following steps: i) write the interface `IA`; ii) write the class `A`; iii) run the `obicomp` tool. The last step is to automatically generate the other interfaces and classes needed, and to extend class A implementing interfaces IProvider and IDemander. Currently, `obicomp` uses a mix of: i) Java reflection mechanism to analyze classes and generate the corresponding proxies, and ii) source code insertion to augment the classes written by the programmer with the methods that implement interfaces IDemander and IProvider.

3.2 Porting Existing Applications

We also wanted to make the porting of existing applications to OBIWAN a simple process, so that an existing application that was written with no distribution in mind, could be easily transformed into a distributed application with access to the functionality provided by OBIWAN (incremental replication). If an existing application is already distributed by means of RMI, we also want to be able to modify it so that it may use functionality supported by OBIWAN.

For non-distributed applications the porting should be performed in the following manner: from every existing class A, an interface IA representing its public methods can be automatically derived (if it does not exist yet). From this interface, OBIWAN automatically generates interface IARemote and classes AProxyOut and AProxyIn. Finally, class A is automatically modified as follows: i) its references to instances of other classes that may be incrementally replicated must be changed to reference the corresponding interfaces so that, by polymorphism, AProxyOut can stand for an instance of A; ii) class A must implement interfaces IA, IDemander and IProvider; the implementation of IA is obvious, and the implementation of IDemander and IProvider is done automatically by `obicomp`.

For a distributed application that was developed with the typical RMI-based approach, the modifications required to make it run on top of OBIWAN are rather easy. Taking into account Figure 3, and given class A, interface IARemote, and class IARemoteImpl, it is necessary to perform the following (all done automatically by `obicomp`): i) generate the interface `IA`; this is similar to the interface `IARemote` that has been written by the programmer and is automatically generated by stripping it of any remote-awareness, namely, exceptions; ii) create, from any existing implementation class (e.g. IARemoteImpl), a local class A; iii) proceed as with a non-distributed application previously described.

Thus, for a distributed application that was developed with the typical RMI-based approach, OBIWAN uses a reverse process to strip the application classes of explicit RMI references and then, deals with them as if they were developed without remoteness in mind, generating automatically the code for remote access and replication.

4 Experimental Results

In this section we present two types of experimental results: i) performance measurements comparing LMI (local method invocation) vs. RMI, and ii) incremental replication with and without clustering with varying parameters. All the performance results we obtained in a 100 Mb/sec local area network, connecting Pentium II and Pentium III PCs with 128 Mb of main memory each, running JDK 1.3.

4.1 Performance of LMI vs. RMI

In this section we present performance results comparing RMI to LMI, i.e. local method invocation, on a single object. The time it takes to make a local method invocation is 20 microseconds[4]. A remote method invocation takes 2,8 milliseconds and, obviously, is independent of the object size. In Figure 4 we present the cost of performing several invocations via RMI and LMI for several object sizes. Note that the execution time of LMI includes the cost due to the creation of the replica and to update it back in the master site. We can conclude that:

- the LMI on a replica performs better than RMI for larger number of invocations and for smaller objects;
- with RMI, the object size has no influence on the invocations time; however, this time grows very sharply with the number of invocations;
- for small objects and few invocations, the performance of RMI and LMI is similar; thus, even in this case, the cost of creating a replica and then updating the master replica is comparable.

4.2 Performance of Incremental Replication

In this section we study the performance of incremental replication for several object sizes: 64 bytes, 1024 bytes and 16 Kbytes. We use a list with 1000 objects (all with

[4]This method performs an access to a variable of the object, so it is not an empty method.

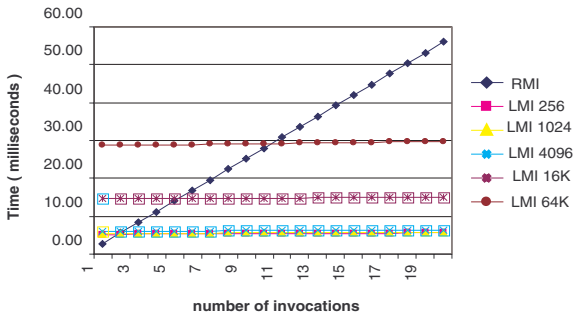

Figure 4. Comparison of RMI and LMI.

the same size) that is created in site S2. This list is then replicated into another site S1, in several steps, each step replicating 1, 25, 100, 250, 500, or 750 objects. Then, the application running in site S1 invokes a 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 Figure 5. Note that, the time values include the creation and transference of all the replicas along with the corresponding proxy-out/proxy-in pairs for each object being replicated. So, in this case, each object still can be individually updated because there is no clustering of objects.

From Figure 5, we can conclude that:

- the steps observed are due to the creation and transference of replicas along with the corresponding proxy-in/proxy-out pairs;
- the creation and transference of replicas along with the corresponding proxy-in/proxy-out pairs is more significant than object invocations;
- the incremental replication of one object each time is the most flexible alternative but is the least efficient for large number of invocations;
- the incremental replication of 25 to 100 objects each time is the most efficient alternative;
- the incremental replication of 750 or 500 objects each time is not efficient because of the high cost of creation and transference of the corresponding replicas and proxy-out/proxy-in pairs;
- for info-appliances with reduced 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 each time).

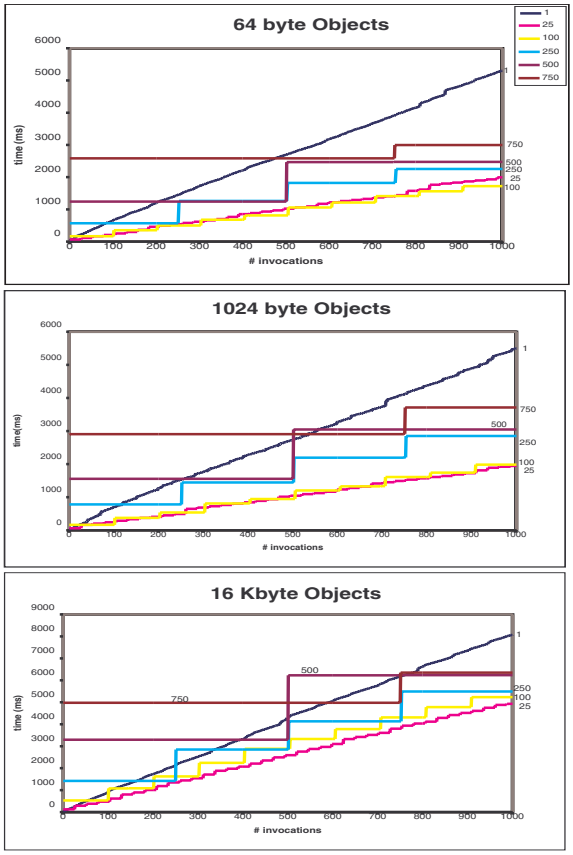

Figure 5. Incremental replication of objects.

4.3 Performance of Cluster Replication

In this section we study the performance of incremental replication with object clustering. The list and object sizes are the same of the previous section. The application running in site S1 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 of these clusters, all objects are replicated as a whole, thus there is only one proxy-in/proxy-out pair being created. Consequently, each object can not be individually updated.

The results are presented in Figure 6. Note that, in each case, the time values include the creation and transference of all the replicas along with the single corresponding proxy-out/proxy-in pairs.

From Figure 6, we can conclude that:

- when compared to the previous section the performance results are much better because there is only one proxy-out/proxy-in pair being created and transferred for each cluster; so, the most significant performance

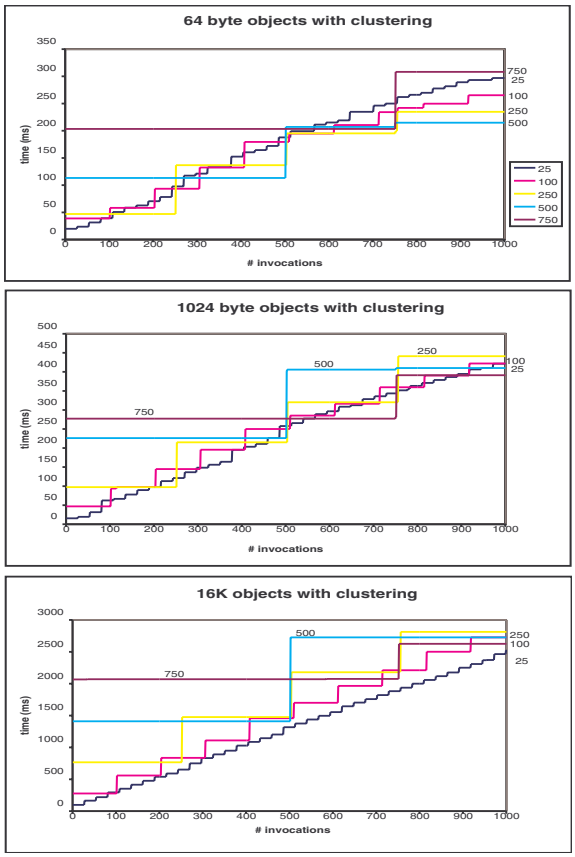

Figure 6. Incremental replication of clusters of objects.

cost is data serialization (done by the Java virtual machine) and network communication;

- 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); the reason is the same as in the previous item.

5 Related Work

The OBIWAN platform is related to several other systems that support distributed invocation, replication, automatic detection and resolution of object-faults. An important difference is that all these systems do not provide an integrated platform supporting all the mechanisms as OBIWAN does. This integration is an advantage to the programmer as he may decide what functionality is best adapted to his application scenario.

Javanaise [2, 11] 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. Javanaise clusters are defined by the programmer and are less dynamic than in OBIWAN. In other words, the frontier of the clusters in OBIWAN are defined in run-time by the application in order to improve its performance and to allow disconnected work.

There has been some effort in the context of CORBA to provide support for replicated objects [10]. There has been also similar efforts in the context of the World Wide Web [2]. However, most of this work addresses other specific issues such as group communication, replication for fault-tolerance, protocols evolution, etc. None seems to address the issue of distributed application development for networks of info-appliances with mobility in mind. OBIWAN attempts to minimize bandwidth and connection time to address this issue. With OBIWAN, the programmer has the means to make his application to decide, in run-time, if an object should be invoked via RMI or if a local replica should be created. We believe that this is a very important aspect when developing distributed applications for info-appliances given the significant and rapid changes in the quality of service of the underlying network.

The issue of object caching has been addressed by many systems. This is different from what we propose, replication: in OBIWAN objects can be replicated freely among sites. However, there are some common aspects between caching and replication. An important distributed system with object and page caching is Thor [14]. This system provides a hybrid and adaptive caching mechanism handling both pages and objects. In addition, Thor provides its own programming language. Most object-oriented databases (OODBs) [22], for example such as O_2 [7], GemStone [1], do have some kind of caching. However, they are very heavyweight, and often come with their own specialized programming language. There has been some work on object caching in CORBA as well. For example, Chockler [8] proposes an hierarchical cache system in which servers cache objects that clients will then ask for. When compared to OBIWAN, it is clear that we do allow objects to be replicated in the client and there is no hierarchical caching system.

Much work has been done regarding object-fault handling [12, 21]. However, most of it has been centered on persistent programming languages or related to adding transparent, orthogonal persistence to existing programming languages. Nevertheless, it is useful, since it introduces well-known and widely accepted designations for relevant existing techniques and/or concepts, e.g. swizzling. Our object-fault handling is done without modifying the underlying virtual machine. This makes our solution more

portable.

6 Conclusions

In wide area networks, distributed applications must be capable of dealing with variable quality of service and disconnections. The mechanism of object replication supported in OBIWAN allows the programmer to deal with such situations; applications may decide, at run-time, what is the best way to invoke an object: via remote method invocation (RMI), or locally via local method invocation (LMI) based on a replication mechanism that brings objects to the info-appliance where an application is running.

The flexibility of the invocation mechanism allows the application programmer to develop his application in such a way that the user can continue to work disconnected from the network (either voluntary or not). As long as the objects needed by an application (or an agent) are locally accessible, there is no need to be connected to the network. In addition, by replicating objects in the info-appliance where an application using them is running, the overall performance can be improved w.r.t. an approach in which objects are always invoked via RMI.

We showed how incremental replication and object faulting resolution can be done without modifying the Java virtual machine. We think that a similar solution could be used for other virtual machines (e.g., .Net [16]). The number of objects being replicated can be changed in run-time by the application. This allows the application to balance latency, bandwidth, invocation performance and memory used. All these aspects are of utmost importance in a mobile wide-area network of info-appliances. The performance results of our first prototype, even with no special optimizations, are very encouraging.

We plan to test our prototype on several info-appliances under different network conditions (wide-area and wireless). We will study how the performance numbers presented in Section 4 depend on the relative speed of the processors involved, for example, between a hand-held PC such as Compaq iPAQ, and a desktop PC. In the future, instead of inserting code (with `obicomp`) into classes written by the programmer, we plan to use a byte-code manipulation library, such as Javassist [4], JOIE [5] or BCEL [6], to avoid source code dependency. Additionally, we want to investigate the possibility of expressing our compiler and code augmenter in declarative terms by making use of results in work like Kava [20].

References

[1] P. Butterwoth, A. Otis, and J. Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, Oct. 1991.

[2] S. J. Caughey, D. Hagimont, and D. B. Ingham. Deploying distributed objects on the internet. *Recent Advances in Dist. Systems, Springer Verlag LNCS, Eds. S. Krakowiak and S.K. Shrivastava*, 1752, Feb. 2000.

[3] D. Chappell. *Understanding ActiveX and OLE*. Redmond, WA: Microsoft Press, 1996. ISBN 1-572-31216-5.

[4] S. Chiba. Javassist — a reflection-based programming wizard for java. In *Proc. of OOPSLA'98 W'shop on Reflective Programming in C++ and Java*, Oct. 1998.

[5] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with joie. In *Proc. of the 1998 USENIX Annual Technical Symposium*, 1998.

[6] M. Dahm. Byte code engineering with the javaclass api. Technical report b-98-17, Freie Universitt Berlin, Institut fr Informatik, 1998.

[7] O. Deux et al. The O_2 system. *Comm. of the ACM*, 34(10):34–48, Oct. 1991.

[8] G. C. el al. 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, Apr. 2000.

[9] R. Englander and M. Loukides. *Developing Java Beans*. O'Reilly & Associates, 1997. ISBN: 1565922891.

[10] P. Felber, R. Guerraoui, and A. Schiper. Replication of corba objects. *Recent Advances in Dist. Systems, Springer Verlag LNCS, Eds. S. Krakowiak and S.K. Shrivastava*, 1752, Feb. 2000.

[11] D. Hagimont and F. Boyer. A configurable rmi mechanism for sharing dist. java objects. *IEEE Internet Computing*, 5, Jan. 2001.

[12] A. L. Hosking and J. E. B. Moss. object fault handling for persistent programming languages: a performance evaluation. In *ACM Conf. on Object-Oriented PRogramming Systems, Languages and Applications, 288-303*, Sept. 1993.

[13] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.

[14] B. Liskov, M. Day, and L. Shrira. Dist. object management in Thor. In *Proc. Int. W'shop on Dist. Object Management*, pages 1–15, Edmonton (Canada), Aug. 1992.

[15] M. W. Oliphant. The mobile phone meets the internet. *Software Practice and Experience*, 36(8):20–28, Aug. 1999.

[16] D. S. Platt. *Introducing Microsoft .Net*. Microsoft Press, 2001. ISBN: 0-7356-1377-X.

[17] J. M. Reuter. *Inside Windows CE*. Microsoft Programming Series. Microsoft Press, 1998. ISBN 1-57231-854-6.

[18] M. Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proc. of the 6th Intl. Conf. on Dist. Systems*, pages 198–204, Boston, May 1986.

[19] B. Venners. *Inside the Java Virtual Machine*. Java Masters Series. McGraw-Hill, 1997. ISBN 0079132480.

[20] I. Welch and R. J. Stroud. Kava using byte code rewriting to add behavioural reflection to java. In *Proc. of the Sixth USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio (USA), Jan. 2001.

[21] S. J. White and D. J. Dewitt. A performance study of alternative object faulting and pointer swizzling strategies. In *18th VLDB Conf. Vancouver, British Columbia, Canada*, 1992.

[22] S. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems*. Morgan-Kaufman, San Mateo, California (USA), 1990.