Distributed Object Invocation in OBIWAN*

Luís Veiga João Garcia João Silva Paulo Ferreira
{luis.veiga,joao.c.garcia,joao.n.silva,paulo.ferreira}@inesc.pt
INESC/IST, Rua Alves Redol Nº 9, Lisboa, Portugal
http://www.gsd.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 an environment in which network connections are slow and not reliable.

For this purpose, we developed a platform called OBIWAN that: i) allows the application programmer to decide the mechanism by which objects should be invoked, remote method invocation or invocation on a local replica, and ii) provides hooks for the application programmer to implement a set of application specific properties such as transactional support, for example.

This functionality allows the application programmer to deal with situations that frequently occur in a wide-area network, such as disconnections and slow links. As a matter of fact, as long as the objects needed by an application are locally accessible, there is no need to be connected to the network. In addition, it allows the programmer to easily replace, in run-time, remote by local invocations, thus improving the performance of his application and its adaptability.

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

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 teaching community, a virtual enterprise grouping several companies from different countries, 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) is 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. This is confirmed by the existence of operating systems and virtual machines for such devices; for example, Windows CE [11] for a number of PDAs currently in the market, and Java for PalmPilot VII [12]. In addition, the foreseen increase of bandwidth in wireless communication makes the connection of these info-appliances to the Internet a reality [10].

We envisage a general scenario in which a user will want 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 partitions).

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.

1.1 Environment

Wide area fully wired computing environments are characterized by a fair amount of bandwidth and low latency when compared to wireless networks. However, when compared to local area networks, the bandwidth

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

is not that big and latency can be clearly noticeable. In addition, partitions do occur making most distributed applications to malfunction.

Thus, we can say that the quality of service we observe in wide area networks is acceptable for most used current applications (such as web browsers) as long as there are no partitions and the network is not too loaded. The problem is that partitions and overload do occur and, as a result, a disruptive latency is frequently observed when accessing, for example, a remote web server. Complex applications, i.e. those effectively supporting collaboration among different users with a strong need for accessing shared data simply stop working when the quality of service degrades.

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

On the other hand, mobile computing is characterized by significant and rapid changes in its supporting infrastructure and, in particular, in the quality of service available from the underlying communication channels; wireless links provide lower bandwidth, possibly higher error rates than wired networks, and periods of disconnection and intermittent or variable connectivity may occur.

Network partitions are rare in stationary local area networks; on the contrary, they occur in greater number in wide area mobile networks. Most applications consider them to be failures that are exposed to users. In the mobile environment, applications will face frequent, lengthy network partitions. Some of these partitions 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 partitions 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 copies of global data.

1.2 OBIWAN

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

We believe that the notion of a generic object broker infrastructure provides the means to the kind of sharing described above. Intuitively, to describe our object broker infrastructure, we can say that OBIWAN supports applications that manipulate an ocean of objects; these objects are scattered over a variety of locations and info-appliances, can flow among such appliances, and contain innumerous references connecting them.

More precisely, OBIWAN provides support for such objects in the sense that they can be invoked either remotely, via remote method invocation (RMI) [1, 13], or locally via local method invocation (LMI) based on a replication mechanism that brings objects to the info-appliance where an application is running [7].

This flexibility of the invocation mechanism allows the application programmer to develop his application so that it resists to network failures, and allows the user to work disconnected from the network (either voluntary or not). As a matter of fact, as long as the objects needed by an application 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. In this paper we focus on this issue, i.e. invocation of distributed objects in OBIWAN.

This paper is organized as follows. In the next section we present the design and implementation of OBIWAN focusing on the support for object replication. In Sections 3 and 4 we present some performance results and related work, respectively. Finally, in Section 5 we present some conclusions and plans for future work.

2 Object Invocation

As a basic functionality, OBIWAN allows the application programmer, if he wants so, to control which objects should invoked remotely or locally. Obviously, the programmer can simply provide hints to the system and let it decide what the best option is. In addition, the programmer can also control, or simply provide hints to the system so that it can decide, what part of an object should be brought near an application, e.g. the whole object (data and code) or just its data or just its code. We claim that this notion of a generic object broker is mostly adequate to the specific needs of wide area mobile sharing and collaboration support.

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 or remotely (by means of RMI).

An object that is invoked locally can be a replica of a master in some remote process; in this case, the master replica can still be invoked via RMI (see Figure 1). So, at any time, both replicas, the master and the local,

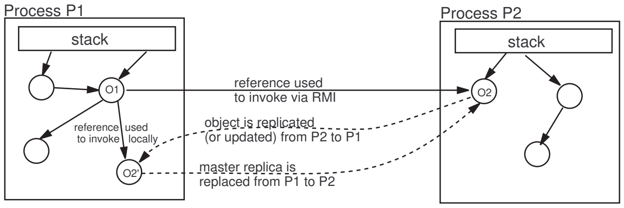

Figure 1: Object O2 in P2 is the master replica; O2' in P1 is a replica that can be invoked locally.

can be freely invoked. It is the programmer who decides what the best option is.

The local replica can be updated from the master whenever the programmer wants; conversely, the local replica can replace the master whenever the programmer wants. There is no limit to the number of replicas.

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. However, OBIWAN may provide a set of classes implementing well-known consistency policies, ready to be used by the programmer.

The OBIWAN system is a software layer on top of a virtual machine. This virtual machine must support distributed object-oriented applications by means of RMI and dynamic code loading.

2.1 Implementation

The OBIWAN system runs on top of the Java virtual machine. For us, this was the obvious choice given that it is portable, free, simple to use, and supports the basic functionality required, i.e. RMI and dynamic code loading.

OBIWAN is a set of interfaces, an automatic code generator to help the application programmer, and a methodology for programming distributed applications. The programming of a distributed application with OBIWAN, when compared to a standard approach based on RMI, is simpler. As a matter of fact, the programmer only has to write: i) an interface specifying the methods that an object will service, and ii) a class implementing the just mentioned interface. All the rest, i.e. the code handling the invocations either via RMI or LMI (after the creation of a local replica), is automatically generated with a tool similar to `rmic`. We illustrate this process with an example.

2.1.1 Classes and Interfaces

Suppose we want to build a distributed application with an object that can be either locally replicated or invoked via RMI. All the interfaces and classes involved

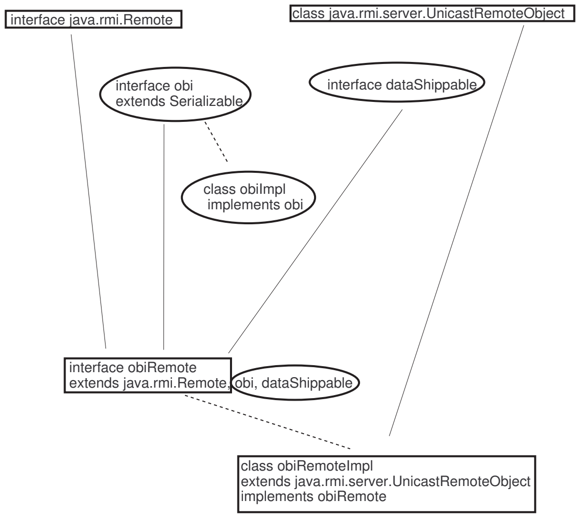

Figure 2: Interfaces and classes of OBIWAN. Inheritance is represented with a solid line; implementation is represented with a dashed line.

```
import java.io.*;
import java.util.*;
import java.rmi.*;
public interface obiInterface extends Serializable {
   public void setVal(int val) throws RemoteException;
   public int getVal( ) throws RemoteException;
}
```

Figure 3: Interface `obi` written by the programmer.

in this example are illustrated in Figure 2. The "rectangular" interfaces and classes are those that can be found in any typical RMI-based client-server application; the "ellipse" interfaces and classes are specific to OBIWAN.

For simplicity, let's say that this object supports the service of a counter; thus, it provides two obvious methods: `setVal(int)` and `getVal()`. This functionality is specified by the interface `obi` and implemented by the class `obiImpl` (see Figures 3 and 4, respectively).

The client program, i.e. the code that invokes the above mentioned object (an instance of the class `obiImpl`) is shown in Figure 5. The code of the server process is shown is Figure 6.

The interface `dataShippable` is constant, thus it does not have to be generated each time an application is programmed. The interface `obiRemote` and the class `obiRemoteImpl` are generated automatically; a part of the generated Java code is the same for any application (for example, the methods `getData` and `putData`) and another part results directly from the interface `obi`.

To summarize, when a new application is developed the programmer does the following steps: i) write the interface `obi`, ii) write the class `obiImpl`, and iii) run the `obicomp` tool to automatically generate the other

```
import java.rmi.*;
import java.io.*;
public class obiImpl implements obi {
   private int count;
   public obiImpl() {count = 0;}
   public void setVal(int x) throws RemoteException {count = x;}
   public int getVal() throws RemoteException {return count;}
}
```

Figure 4: Classe `obiImpl` written by the programmer.

```
public static void main(String [] args) {
   String host;
   if (args.length > 0) host = args[0] else host="localhost";
   System.setSecurityManager(new RMISecurityManager());
   obiRemote refRMI =
      (obiRemote)Naming.lookup("//"+host+"/obi");
   if (refRMI==null) throw new Exception("RMI failed!!!");
   obi refLMI = (obi) refRMI.getData();
   if (refLMI == null) throw new Exception("LMI failed!!!");
   System.out.println("Remote value" + refRMI.getVal());
   System.out.println("Local value" + refLMI.getVal());
   refLMI.setVal(333);
   refRMI.setVal(444);
   System.out.println("Remote value " + refRMI.getVal());
   System.out.println("Local value " + refLMI.getVal());
   refRMI.putData(refLMI);
   System.out.println("Remote value " + refRMI.getVal());
   System.out.println("Local value " + refLMI.getVal());
}
```

Figure 5: The client code written by the programmer.

```
public static void main(String args[]) {
   System.setSecurityManager(new RMISecurityManager());
   try {
      obiRemoteImpl server = new obiRemoteImpl();
      Naming.rebind("obi", server);
   } catch (java.io.IOException e) {
      System.out.println("// Problem registering server");
      System.out.println(e.toString());
      }
   }
```

Figure 6: The server code written by the programmer.

```
import java.util.*;
import java.rmi.*;
public interface dataShippable {
   public Object getData() throws RemoteException;
   public void putData (Object replica) throws RemoteException;}
}
```

Figure 7: Interface `dataShippable` generated automatically.

```
import java.rmi.*;
public interface obiRemote
extends java.rmi.Remote, obi, dataShippable{}
}
```

Figure 8: Interface `obiRemote` generated automatically.

```
import java.rmi.*;
import java.util.*;
public class obiRemoteImpl
   extends java.rmi.server.UnicastRemoteObject
   implements obiRemote {
   obiImpl data;
   public obiRemoteImpl() throws RemoteException {
      data = new obiImpl();
   }
   public void setVal(int x) throws RemoteException {
      data.setVal(x);
   }
   public int getVal() throws RemoteException {
      return data.getVal();
   }
   public Object getData() throws RemoteException {
      return data;
   }
   public void putData (Object replica) throws RemoteException {
      this.data = (obiImpl) replica;
   }
}
```

Figure 9: Class `obiRemoteImpl` generated automatically.

interfaces and classes needed.[1]

2.2 Porting Existing Applications

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 2, it is necessary to perform the following source code modifications.

1. Generate the interface `obi`; this is similar to the interface `obiRemote` that has been written by the programmer.
2. Generate the class `obiImpl` that is similar to the class `obiRemoteImpl` that has been written by the programmer.
3. Modify the interface `obiRemote` so that it extends the interfaces `obi` (generated automatically in the first step) and `dataShippable`, and contains no methods.
4. Modify the class `obiRemoteImpl`: instance variable of type `obiImpl`, methods `getData` and `putData`, and all the methods of the interface `obiRemote` that simply invoke the instance variable previously mentioned. Thus, the class `obiRemoteImpl` behaves as a wrapper of class `obiImpl`.

We are currently developing a tool that performs all these operations automatically. However, there is something that is not easy to automate: the kind of invocations done by the client, i.e. RMI or LMI on a replica. This decision is application specific and the application programmer is most probably the best person to do it.

[1] The `obicomp` tool makes use of `rmic` and uses reflection to generate code automatically.

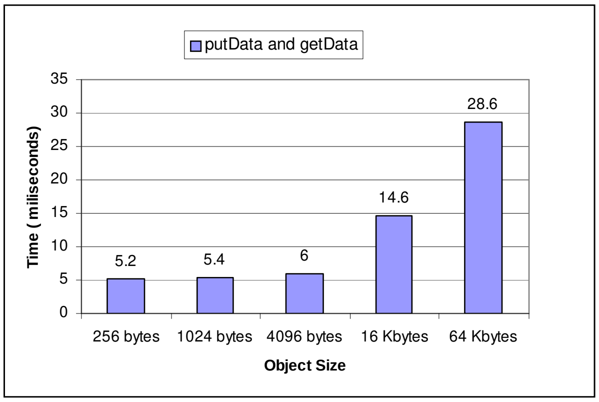

Figure 10: Cost of creating a replica (`getData`) plus updating the master replica (`putData`).

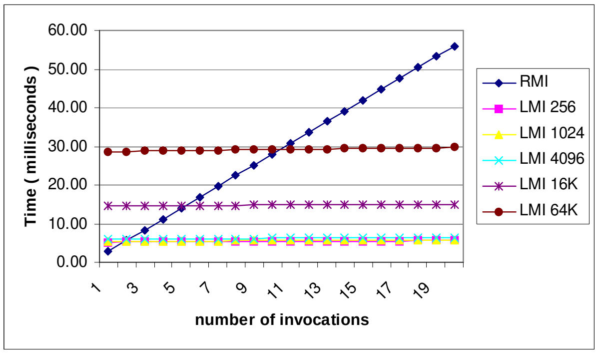

Figure 11: Comparison of RMI and LMI.

3 Performance

In this section we present some preliminary performance results comparing RMI to LMI on a replica. The values were obtained in a 100 Mb/sec local area network, connecting two Pentium II PCs with 64 Mb of main memory each.

The time it takes to make a local method invocation is 20 microseconds[2]. A remote method invocation takes 2,8 milliseconds and, obviously, is independent of the object size. The cost of loading the object (code and data) when a local replica is created (method `getData`), and to update the master replica once all the local invocations have been performed (method `putData`) is presented in Figure 10; as expected is depends on the object size.

In Table 1 we present the number of invocations above which the creation of a replica and its local invocation performs better than RMI. In Figure 11 we present the cost of performing several invocations via RMI and LMI. From Table 1 and Figure 11 we can conclude that:

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

object size	number of invocations
256 bytes	1,9
1024 bytes	1,9
4096 bytes	2,2
16 Kbytes	5,3
64 Kbytes	10,3

Table 1: Number of invocations above which the creation of a replica and its local invocation performs better than RMI.

- 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 are similar; thus, even in this case, the cost of creating a replica and updating the master replica is comparable.

4 Related Work

Javanaise [2] aims at providing support for cooperative distributed applications on the internet. This system is more complex than ours in the sense that it addresses more issues, such as consistency, object clustering, etc. 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.

There has been some effort in the context of CORBA to provide support for replicated objects [4]. The same applies to the web [3]. 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.

OBIWAN is a system simpler than those previously mentioned. The basic difference is that it does not try to provide transparency, i.e. the application programmer does not develop his application as if it was centralized; he does know that his application is distributed. 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.

5 Conclusion and Future Work

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 that 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 so that it resists to network failures, and allows the user to work disconnected from the network (either voluntary or not). As long as the objects needed by an application 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. A first prototype was built in Java and the performance results obtained are 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 3 depend on the relative speed of the processors involved, for example, between a hand-held PC such as HP Jornada 820, and a desktop PC.

We are also working on improving the functionality attached to the methods `getData` and `putData` in order to provide transactional support. Basically, these methods will implement begin-transaction and end-transaction, respectively. The challenge is to provide relaxed concurrency policies well adapted to the environment under consideration (see Section 1.1) [9]. In this area, we intend to take advantage of our previous work in the PerDiS project [7, 8].

We also intend to study the impact of the OBIWAN approach on garbage collection. As a matter of fact, the distributed collector in Java associates leases to inter-process references; thus, if the lease associated to the reference for a master replica expires (e.g. from O1 to O2 in Figure 1) this may lead to the reclamation of the master replica; however, a replica still exists. This situation arises because the Java distributed collector is not aware of replicas; for the collector, these are simply different objects. Once again, we plan to extend our previous work in this area [5, 6].

References

[1] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 217–230, 1993.

[2] Steve J. Caughey, Daniel Hagimont, and David B. Ingham. Deploying distributed objects on the internet. In S. Krakowiak and S.K. Shrivastava, editors, *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*. Springer Verlag, 2000.

[3] Steve J. Caughey, Daniel Hagimont, and David B. Ingham. Deploying distributed objects on the internet. In S. Krakowiak and S.K. Shrivastava, editors, *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*. Springer Verlag, 2000.

[4] Pascal Felber, Rachid Guerraoui, and André Schiper. Replication of corba objects. In S. Krakowiak and S.K. Shrivastava, editors, *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*. Springer Verlag, 2000.

[5] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–241, Monterey CA (USA), November 1994. ACM. http://www-sor.inria.fr/SOR/docs/GC-DSM-CONSIS_OSDI94.html.

[6] Paulo Ferreira and Marc Shapiro. Modelling a distributed cached store for garbage collection: the algorithm and its correctness proof. In *ECOOP'98, Proc. of the Eight European Conf. on Object-Oriented Programming*, Brussels (Belgium), July 1998.

[7] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, João Garcia, Sytse Kloosterman, Marcus Roberts Nicolas Richer, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. PerDiS: design, implementation, and use of a persistent distributed store. In S. Krakowiak and S.K. Shrivastava, editors, *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*. Springer Verlag, 2000.

[8] João Garcia, Paulo Ferreira, and Paulo Guedes. The perdis fs: A transactional file system for a distributed persistent store. In *Proc. of the 8th ACM SIGOPS European Workshop*, Sintra, (Portugal), September 1998.

[9] Qi Lu and M. Satyanarayanan. Improving data consistency in mobile computing using isolation-only transactions. In *Proceedings of the 5th Workshop on Hop Topics in Operating Systems*, Orcas Island (WA, USA), May 1995.

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

[11] John Muray Reuter. *Inside Windows CE*. Microsoft Programming Series. Microsoft Press, 1998. ISBN 1-57231-854-6.

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

[13] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. In *Conference on Object-Oriented Technologies*, Toronto Ontario (Canada), 1996. Usenix.