UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Memory Management in Ubiquitous Computing

(Gestão de Memória em Computação Ubíqua)

Luís Manuel Antunes Veiga

(Mestre)

Proposta de Dissertação e
Plano de Estudos para obtenção do Grau de
Doutor em Engenharia Informática

Orientador:	Doutor Paulo Jorge Pires Ferreira
Comissão:	Doutor José Augusto Legatheaux Martins
	Doutor José Manuel da Costa Alves Marques
	Doutor Arlindo Manuel Limede de Oliveira
	Doutor Paulo Jorge Pires Ferreira

Maio 2003

O presente documento consiste na proposta de tema de dissertação e plano de estudos intitulado “Memory Management in Ubiquitous Computing (Gestão de Memória em Computação Ubíqua)” e foi escrito por Luís Manuel Antunes Veiga para apreciação da Comissão de Acompanhamento dos Trabalhos de Doutoramento em Engenharia Informática e de Computadores, no Departamento de Engenharia Informática, do Instituto Superior Técnico, da Universidade Técnica de Lisboa.

O Plano de Estudos encontra-se, naturalmente, redigido em português. Quanto ao conteúdo da proposta de tese, está em língua inglesa e segue a linha das submissões já publicadas no decurso do Doutoramento. Contempla, também, a submissão realizada à IEEE Transactions on Parallel and Distributed Systems, com as naturais adendas versando, com algum detalhe, os vários aspectos por aprofundar e que ainda não foram alvos de submissão.

Este trabalho tem sido realizado no âmbito do projecto OBIWAN em parceria com a Microsoft Research em Cambridge.

Os trabalhos de Doutoramento têm sido realizados sob a orientação do Professor Doutor Paulo Jorge Pires Ferreira, Professor Auxiliar do Departamento de Engenharia Informática, do Instituto Superior Técnico, da Universidade Técnica de Lisboa.

Enquanto que as redes constituídas por computadores estacionários têm topologias estáticas que permitem a introdução de forma simples de mecanismos de configuração, administração e partilha de dados, as redes constituídas por dispositivos móveis de computação surgem com uma índole mais dinâmica.

Na vertente da concepção de mecanismos de suporte às aplicações (sistemas operativos, middleware), apresentam-se com maior exigência, neste novo ambiente variável, questões de adequação, coerência e adaptabilidade no que concerne o ambiente de execução de aplicações.

Neste trabalho abordamos três questões específicas: suporte à execução de aplicações distribuídas, reciclagem de memória distribuída clássica e gestão de memória num sentido mais vasto.

O primeiro ponto visa o suporte à execução e adaptação de aplicações distribuídas aos novos ambientes de computação mencionados anteriormente. Neste contexto, é necessário suportar vários paradigmas de invocação distintos em tempo de execução, incluindo invocação local em réplicas, invocação remota e através de agentes móveis. A replicação incremental de grafos de objectos e o tratamento de faltas de objectos em sistemas distribuídos integrando redes ad-hoc, constituem um suporte fundamental para a computação distribuída.

O segundo ponto consiste no estudo da reciclagem automática de memória distribuída, com e sem replicação, realizada sobre sistemas distribuídos integrando redes ad-hoc. A estes algoritmos, na sua correcção e funcionamento, impõem-se restrições não exigidas aos algoritmos actualmente existentes.

Um prossecução natural destes pontos será o esforço de abordagem das questões levantadas a um nível de abstracção mais elevado e que permita a configuração, automática e adaptativa, do suporte de execução a diferentes dispositivos e a sempre diferentes configurações de recursos numa rede ad-hoc.

Palavras chave:

middleware, replicação, reciclagem automática de memória distribuída, mobilidade.

While networks consisting of desktop computers have fixed and static topology that ease configuration, administration and data sharing, networks consisting of mobile devices have an intrinsic dynamic nature.

Regarding mechanisms to support application execution (operating systems, middleware), issues like adequacy, coherence and adaptability rise with greater importance in this variable environment.

In this work, we deal with three specific issues: support for execution of distributed applications, distributed garbage collection and memory management in a broader sense.

The first one aims to support execution and adaptation of distributed applications to the new environments consisting of the devices mentioned earlier. Therefore, incremental replication of graphs of objects and object-fault handling in distributed systems built over ad-hoc networks, are a fundamental support to distributed computation.

The second point addresses distributed garbage collection, with and without replication in mind, in distributed systems over ad-hoc networks. Current algorithms must be extended in correctness, and improved in performance, to deal with new constraints imposed by these systems.

A natural follow-up of these subjects will be the effort to address these issues, at a higher level of abstraction, that allows automatic adaptation and configuration of the execution environment, to different devices and amount of resources in these systems.

Keywords:

middleware, replication, distributed garbage collection, mobility.

1

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. object replication, distributed garbage collection).

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).

Currently, dealing which such paradigm diversity implies: i) either using different middleware platforms, with obvious conveniences such as integration problems and learning costs, or ii) when sticked to one middleware platform the programmers are forced to deal with system-level issues such as handling the creation of replicas and the corresponding consequences in terms of object faulting, among other details.

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

- **paradigm flexibility** - allows the programmer to develop his applications using either RMI, object replication, or mobile agents[2] according to the specific needs of applications;
- **automatic replication** - supports distributed memory management that is capable of dealing with object replicas automatically (e.g., incremental on-demand replication, transparent object faulting and serving, etc.);

[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.

- **support for mobile agents** - support for migration of execution flows.
- **distributed garbage collection** - supports the automatic reclamation of useless replicas;

We would like to emphasize the fact that no other middleware platform provides all the characteristics mentioned above. In particular, the support for automatic replication raises the problem of distributed memory garbage collection more seriously than with traditional RMI; garbage collection solutions for distributed systems based on RMI are not safe when applied to a distributed system with replicas. In addition, it's equally important to note that the usage of mobile agents brings the problem of abusive resource consumption on the hosting computers. OBIWAN solves this problem by means of a security language and a monitor that enforces the policies thus defined (not necessarily by the programmer (Dias et al. 2003; Ribeiro et al. 2001)). This latter subject, security, is not included in the thesis contributions and will not be addressed in detail in this document.

Concerning replication, it rises issues regarding consistency of different replicas in the presence of independent updates. These are very important issues, which have been addressed by past(Liskov et al. 1992; Demers et al. 1994) and more recent work(Felber et al. 2000; Caughey et al. 2000; Preguiça et al. 2002; Preguiça et al. 2003). Therefore, OBIWAN provides hooks where consistency policies can be integrated in the system. However the research and implementation of these aspects is out of the scope of this work.

Finally, OBIWAN runs both on top of Java (Arnold and Gosling 1996) and .Net (Platt 2001) 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 is responsible for dealing with all aspects of replica creation so that: i) it allows the application to decide, in run-time, the mechanism by which objects should be invoked, RMI or invocation on a local replica, ii) it allows incremental replication of large object graphs, and iii) 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: i) as long as objects needed by an application (or by an mobile 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.

1.2 Garbage Collection

Concerning distributed garbage collection (DGC), most algorithms (Plainfossé and Shapiro 1995) are not well suited for systems supporting object replication because: either (i) they do not consider the existence of replication, or (ii) they impose severe constraints on scalability by requiring causal delivery to be provided by the underlying communication layer(Ferreira and Shapiro 1994; Ferreira and Shapiro 1996)

In OBIWAN, the garbage collection algorithm solves both these problems. The result is a DGC algorithm that, besides being correct in presence of replicated data 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 Problems to Solve and Expected Contributions of this Thesis

This thesis aims to develop contributions in the area of Memory Management in Ubiquitous (or Pervasive) Computing, focusing mainly on the following subjects:

- Incremental replication of graphs of objects and transparent handling of object-faults in distributed systems, including those based on ad-hoc networks.
- Distributed garbage collection both in the presence and absence of replicated data.
- Memory management modelling, in a broader sense, in ad-hoc networks of devices with limited capabilities. This includes foreseeing and managing de-localization of objects (object-swapping) in order to comply with memory limitations, hiding these from the programmer.

The rest of this thesis proposal is organized as follows. In Chapter 2 we address the state-of-the-art in the related area and compare it to the work already developed in this context. In Chapter 3 we describe the architecture of OBIWAN focusing on its most relevant components: support for incremental replication, mobile agents; and distributed garbage collection. In the Chapter 4 we present the most important aspects of the prototype implementated and in Chapter 5 we show some relevant performance results we already got. In Chapter 6 we draw some preliminary conclusions and introduce on-going and immediate future work.

2

The OBIWAN platform can be related to several other systems that support remote invocation, replication, DGC and mobile agents. 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 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.

2.1 Object-Fault Handling

Much work has been done regarding object-fault handling (Hosking and Moss 1993; Sousa et al. 1993; White and Dewitt 1992; Wilson and Kakkad 1992). 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.

2.2 Replication, Caching, Mobility

Javanaise (Caughey et al. 2000; Hagimont and Boyer 2001) 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 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. 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 (Felber et al. 2000) as well as in the context of the World-Wide-Web (Caughey et al. 2000). However, most of this work addresses other specific issues such as group communication, replication for fault-tolerance, protocols evolution, etc.

The issue of object caching and consistency(Franklin et al. 1997) and has been addressed by many systems. This is different from the replication mechanism we propose: in OBIWAN objects can be replicated freely among sites (not just in client-server fashion) and it does not enforce a specific consistency policy. However, there are various common aspects between caching and replication; therefore, we discuss some related aspects with work done in this area.

Most OODBs (Zdonik and Maier 1990), e.g. O_2 (Deux et al. 1991) and GemStone (Butterwoth et al. 1991) 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.

Thor (Liskov et al. 1992) 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 distributed garbage collector (which does not handle replicas correctly).

Mobisnap(Preguiça et al. 2003) is a database middleware system designed to transparently support applications running on mobile environments. It allows different clients concurrently updating the database by usage of mobile transactions and reservations though modification of persistent data is only done at the central server. Mobile transactions are expressed in unmodified PL/SQL, improving on previous results(Preguiça et al. 2000). It allows caching of relational-model data in the clients and semantically infers from client transactions, the necessary constraints (reservations) which, with a good degree of confidence, prevent conflicts and allow transaction completeness, independently, when these are replayed at the central server.

Pro-Active(Baduel et al. 2002) is a Java library for distributed programming with mobility in mind. It is originally based on a programming model described in detail in (Caromel 1993). Some of its goals are common to ours, namely, to facilitate programmer's lives in dealing with distribution issues. Neither

OBIWAN nor Pro-Active impose any changes on the underlying virtual machines. Both provide weak mobility to support mobile agents migration among different hosts.

However there are several differences: Pro-Active emphasis is on process synchronization, group communication and the absence of specific compilers or source code extenders. OBIWAN does not deal explicitly with synchronization issues and includes such compilers and source code extenders. We provide hooks that programmers can use to implement specific synchronization policies. The OBIWAN platform main design goal is to provide incremental replication (not just migration) through transparent object-fault detection in a distributed environment. Pro-Active does not address distributed garbage collection. OBIWAN uses distributed garbage collection algorithm able to reclaim unreachable replicas of objects. An interesting future work would be to integrate Pro-Active synchronization policies in OBIWAN replication mechanism.

2.3 Distributed Garbage Collection

Previous work in DGC as IRC (Piquer 1991), SSP chains (Shapiro et al. 1992) and Larchant (Ferreira and Shapiro 1998) served as the starting point of the DGC algorithm presented in this work. Our algorithm(Sanchez et al. 2001; Veiga and Ferreira 2003) 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's GC-consistent cuts (Skubiszewski and Porteix 1996). He considers 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 I and II (see Section 3.2.1). The authors prove that this is a sufficient condition for safety and liveness.

The work in (Rodrigues and Jones 1998) is based on the assumption that distributed garbage cycles exist but are less common than acyclic data. The collection of this cycles must be performed more slowly than local or distributed acyclic garbage The algorithm uses a local-tracing collector and distributed reference listing collector, augmented with an incremental three-phase partial trace to reclaim distributed cycles. The first one, mark-red phase identifies a distributed subgraph suspected of being garbage and dynamically identifies groups of processes that will collaborate to reclaim distributed garbage. After this, a scan phase determines whether members of this subgraph are actually garbage and that any other collections upon which this collection depends are finished

(Fessant 2001) presents a detector of free-cycles based on a medium approach between per-cycle and all-at-once detectors. It is based on a distance heuristic, improved by min-max marking so that some suspected cycles are detected earlier. It depends on partial-lazy backtracking mechanism named sub-generation. It provides a global and uniform mechanism with no need for extra messages to detect cycles, It does not require neither consensus, nor object migration and no extra space in objects. It imposes minor modifications on tracing collectors.

(Rodriguez-Rivera and Russo 1997) also makes use of backtracking, instead of mark-sweep, to eliminate the need of global synchronization. Although this is not the the first use of backtracking (e.g. (Maheshwari and Liskov 1995)), it addresses the important issues about implementing this concept in a real environment/system with off-the-shelf software.

2.4 Mobile Agents

There are, today, a great number of Java-based mobile-agent systems available (Silva A. 2001; ObjectSpace 1997; Lange and Oshima 1998; General Magic 1997). All of them share certain characteristics arising from their use of Java as a development platform:

- they all provide an execution environment for the agents, which is the point of contact on a given machine into which agents move and act;
- agents can migrate from server to server carrying their state with them;
- agents can load their code from a variety of sources;
- all platforms are written in pure Java;
- the package of the platform distribution includes a complete documentation system.

Among these, Object Space's Voyager 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, none supports neither objects to be automatically replicated nor provide DGC. In addition, the security specification and enforcement relies solely on native mechanisms (JVM or operating system).

The Aglets Workbench platform from IBM (Lange and Oshima 1998) modulates an agent based on the applet model, which defines a characteristic life cycle, by the implementation of specific methods. Aglets treats agents differently than regular objects and does not allow to send a regular Java message to a moving agent, which makes it very difficult to communicate with an agent and between agents. Furthermore, Aglets uses sockets to move agents, which may complicate platform's portability to other network technology. This platform is very easy to install and use. It is necessary to run Tahiti, the aglet server, which users employ to create, deploy, retract, and kill agents as well as observe the internal server's state.

General Magic's Odyssey (General Magic 1997) is also an agent platform that introduces some interesting mechanisms. Odyssey is quite similar to Aglets in the way it is also very difficult to locate a moving agent. As opposed to Aglets, Odyssey uses RMI to provide agent's migration. Odyssey also has some mechanisms for agent collaboration in order to allow their gathering in specific nodes of the network. However it is very poor in support and available documentation.

All these mobile agent systems allow an execution flow to migrate so that it becomes co-located with its data. However, none supports incremental replication of objects providing a well integrated platform to the programmer as OBIWAN does.

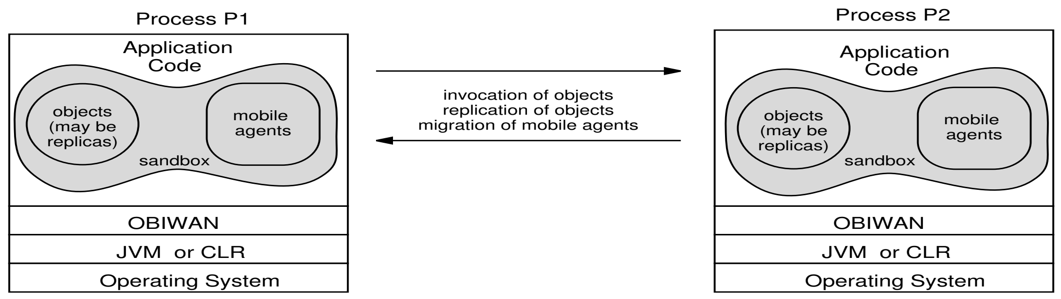
3

OBIWAN is a peer-to-peer middleware platform(Veiga and Ferreira 2001; Veiga and Ferreira 2002a) (see Figure 3.1-a) in the sense that any process may behave either as a client or as a server at any moment. In particular, w.r.t. replication this means that a process P can either request the local creation of replicas of remote objects (P acting as a client) or be asked by another process to provide objects to be replicated (P acting as a server).

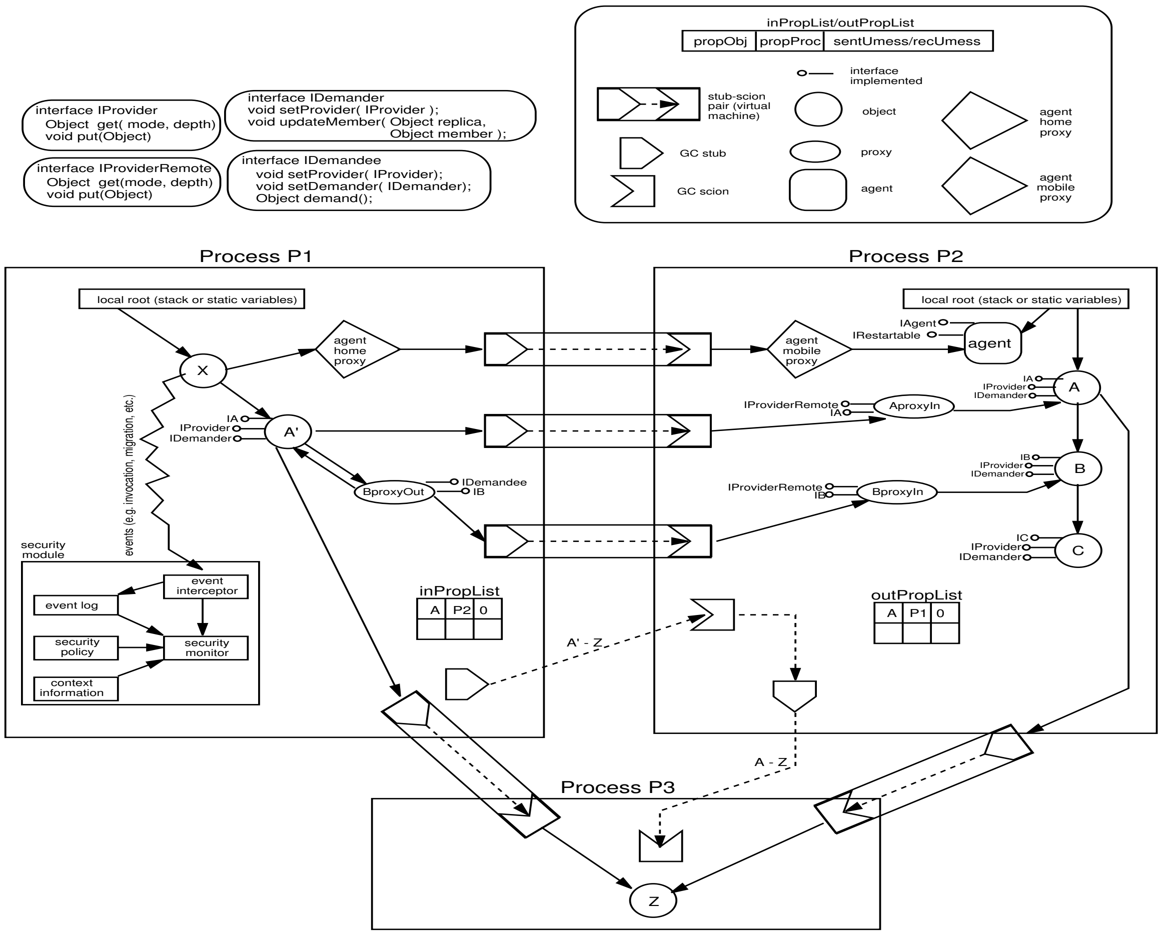
OBIWAN gives to the application programmer the view of a network of computers in which one or more processes run; objects and agents exist inside processes (w.r.t. agents, we call such processes hosts). An object can be invoked locally (after being replicated) or remotely. Mobile agents can be created 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)(Ribeiro et al. 2001). SPL is then automatically translated into code that enforces that policy.

The most important OBIWAN data structures are illustrated in Figures 3.1-b and 3.2:

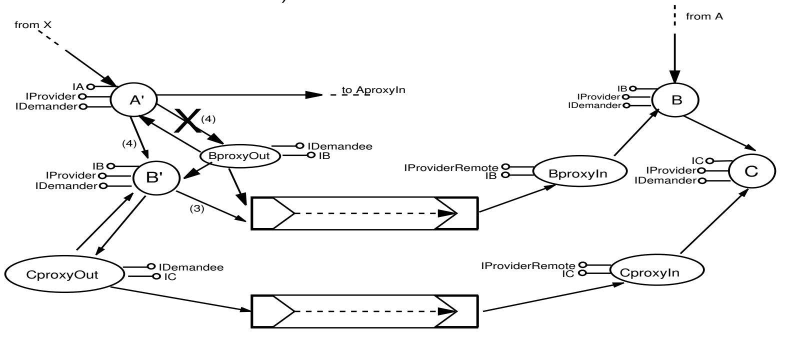
- **Proxy-out/proxy-in pairs** (Shapiro 1986). A proxy-out stands in for an object that is not yet locally replicated (e.g. BproxyOut stands for B' in P1). For each proxy-out there is a corresponding proxy-in. In Section 3.1 we describe how these proxies help in supporting object replication.
- **Interfaces**. The interfaces implemented by each object and proxy-out/proxy-in pairs are the following. **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.
  - **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 (as described in Section 3.1.1).



a) OBIWAN architecture.



b) OBIWAN data structures.



c) Replication of B (numbers close to arrows corresponding to enumerated items in Section 2.1.1).

Figure 3.1: OBIWAN platform: (a) architecture, (b) data structures, and (c) replication of B from P2 to P1.

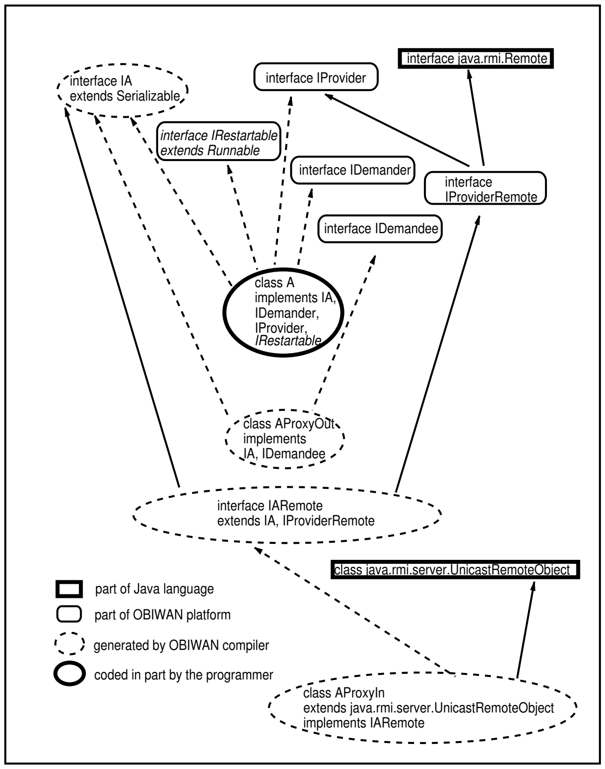


Figure 3.2: Interfaces and classes in OBIWAN

  - **IProviderRemote**: remote interface that inherits from IProvider so that its methods can be invoked remotely.

- **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. Contrarily to the home proxy, the remote one is mobile and travels between hosts, permanently accompanying the agent.

- **GC-stubs and GC-scions**. A GC-stub describes an outgoing inter-process 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 inter-process reference, from a source process to a target process (e.g. to object Z in P3 from object A in P1).

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 Figure 3.1-b) that are managed by the underlying virtual machine.

- **inPropList and outPropList**. These lists indicate the process *from which* each object has been replicated, and the processes *to which* each object has been 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; **sentUmess/recUmess** is a bit indicating if a unreachable message (for DGC purposes) has been sent/received (more details in Section 3.2.1).
- **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 authorization 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 on not.

3.1 Replication

The application programmer, if he wants so, can control, both at compile-time and at run-time, 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.[1]

The incremental replication(Veiga and Ferreira 2002a) 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

[1]Note that the application programmer is not forced to deal with consistency; he may simply use a library of specific consistency protocols.

is smaller because the application can invoke immediately the new replica without waiting for the whole graph to be available, and ii) 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 in run-time, between incremental or transitive closure replication mode, by means of the mode argument of the method IProvideRemote::get(mode,depth).[2]

3.1.1 Incremental Replication

Without loss of generality, we describe how OBIWAN supports replication taking into account the scenario illustrated in Figures 3.1-b and 3.1-c. There are two processes, P1 and P2, in two different sites, and the initial situation is the following: i) P2 holds a graph of objects A, B and C; ii) object A has been replicated from P2 to P1, thus we have A' in P1; iii) A' holds a reference to AproxyIn (for reasons that will be made clear afterwards); iv) 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 afterwards.

The stub-scion pairs for RMI support are created by the underlying virtual machine. Objects A, B and C are created by the programmer; their replicas (A', B', etc.) are created either upon the programmer's request or automatically. 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. invokes method BProxyIn.get in P2 (BProxyIn is BProxyOut's provider);
2. BProxyIn.get invokes B.get (interface IProvider) that will proceed as follows: creates B', CProxyOut, CProxyIn and sets the references between them; once this method terminates, B', BProxyOut

[2]The run-time parameter "depth" specifies the depth of the graph to be replicated.

and CProxyOut are all in P1, CProxyIn is in P2; note that A' and BProxyOut still point to each other (Figure 3.1-c 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's 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 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 a proxy-in/proxy-out pair.

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 processes. Thus, the amount of objects in the cluster can be 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. (See Section 5 for performance results of both approaches.)

3.2 Distributed Garbage Collection

Consider a scenario in which the initial situation is illustrated in Figure 3.1-b. Now, suppose that, due to application execution in P1, A' becomes locally unreachable[3] 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 in Section 3.2.1).

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 (Ferreira and Shapiro 1998) does handle this situation; however, it imposes severe constraints on scalability because it requires the underlying communication layer to support causal delivery (Guerraoui and Schiper 1997). 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.

3.2.1 DGC Algorithm

The DGC algorithm is an hybrid of tracing and reference-listing (Piquer 1991; Shapiro et al. 1992). 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 changing 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 inter-process 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

[3]Locally (un)reachability means (un)accessibility from the enclosing process's local root.

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; 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.

- When a locally reachable object (already discovered by the local collector) is found, the tracing along that reference path ends.
- When an outgoing inter-process reference is found the corresponding GC-stub is created in the new set of GC-stubs.
- 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.
- 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 can not 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: (i) immediately before a message containing a replica is sent, the references being *exported* (contained in the replicated object)[4] must be found in order to create the corresponding GC-scions, and (ii) immediately before a message containing a replica is delivered, the outgoing inter-process references being *imported* must be found in order to create the corresponding local GC-stubs.[5]

It's worthy to note that the DGC algorithm does not require the underlying communication layer to support causal delivery (which is an improvement w.r.t. 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:

I - 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.

II - 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 inter-process references and the corresponding GC-stubs created in P.

III - 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.

3.3 Support for Mobile Agents

To explore and demonstrate the facilities provided in terms of migration of execution flow, a mobile agents platform was implemented, in the context of an undergraduate thesis, based on the OBIWAN framework and supported by OBIWAN compiler and functionality.

This platform is based on a set of independent entities that interact with each other and offer certain services to the programmer.[6] These entities are the agent server that provides the environment for agent execution, the monitor tool that provides information concerning the status of each agent, and the agents themselves.

The programmer is responsible for the implementation of the desired agents. Once the agent's classes are created, they can be instantiated by a programmer's application. Then, the agent is able to

[4]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**.

[5]Note that this may result in the creation of chains of GC-stub/GC-scion pairs, as it happens with SSP Chains (Shapiro et al. 1992).

[6]These services are out of the scope of this thesis.

migrate to a specific process, the Agent server, running locally or in other network nodes. The migration can be triggered by application's explicit order or by the agent himself. The application is able to reach the agent and invoke any of its methods, and may even call it back.

By usage of a name/location directory server, applications can deploy agents able to function in a disconnected fashion. Through an unique agent key, another application started later can still contact and instruct the agent wherever it is located at that moment.

The main results from this work were the following: i) the platform was developed very rapidly because it uses the functionality provided by OBIWAN and its compiler, and ii) the effort done by the programmer to build a set of mobile agents is very small.

To enforce security through the use of obligation polices, SPL(Dias et al. 2003) framework is being implemented over OBIWAN mobile agent support in the context of a M.Sc. thesis.

4

We developed two prototypes of OBIWAN: i) one(Veiga and Ferreira 2002a) runs on top of the JVM and is written in Java, and ii) the other(Veiga and Ferreira 2002b) runs on top of the .Net CLR and is written in C#.[1] 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.

4.1 Classes and Interfaces

The most relevant interfaces and classes concerning replication are illustrated in Figure 3.2 (recall Figure 3.1-b for the corresponding methods). The differences between Java and C# are minimal. Thus, hereafter, we will consider only the Java interfaces.

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 **obicomp**.[2] Finally, the solid "ellipse" represents the class that the programmer writes. The programmer only has to worry with the so-called application-logic.

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.

[1]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#.

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

OBIWAN provides support for the migration of execution flow through the interface IRestartable that is automatically implemented by obicomp. Programmers 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 re-activate it in another process through the creation of a new dedicated object thread. Prior to invoking the checkpoint method, it is the programmer's responsibility to set the object in a stable state that does not rely on stack frame information, i.e. the object can be re-started 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: i) write the interface IA; ii) write the class A; iii) 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 to implement the interfaces Runnable (provided by Java) and IRestartable (provided by OBIWAN); OBIWAN generates automatically the code that implements IRestartable. (Obviously the programmer has to write method run.)

Currently, obicomp uses a mix of: i) reflection 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, IProvider and IRestartable.

4.2 Distributed Garbage Collection

Basically, the code of the distributed garbage collector implements the safety rules (recall Section 3.2.1). 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 implies that the new set of GC-stubs, resulting from a local collection, 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.

5

We intend to fully evaluate the OBIWAN middleware platform by developing new applications, porting existing ones, and measuring its performance. Currently, we only show some relevant performance results of OBIWAN concerning its core functionality: i) the cost of incremental object replication with and without clustering and; ii) the performance penalty due to DGC safety rules.

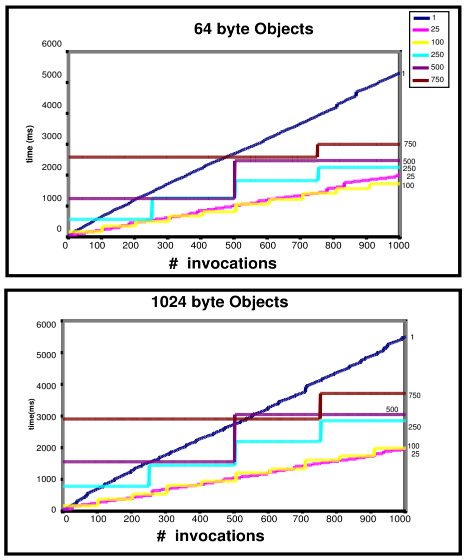
All the results were obtained in a 100 Mb/sec local area network, connecting several PCs with Pentium II and Pentium III processors, either with 64 Mb or 128 Mb of main memory each, running JDK 1.3 on top of Windows 2000.

5.1 Incremental Replication

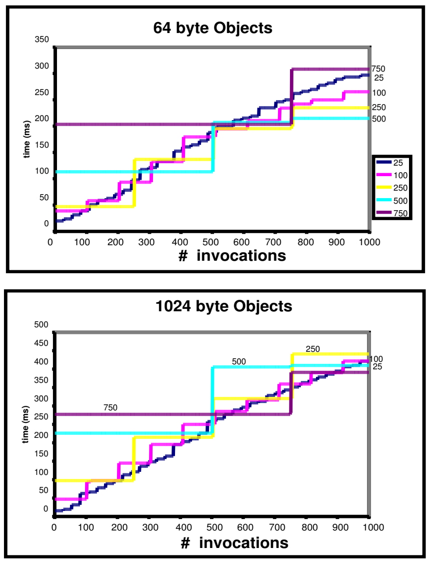
We present the performance of incremental replication for objects with 64 bytes and 1024 bytes. We use a list with 1000 objects (all with the same size) that is created in process P2. This list is then replicated into another process P1, in several steps, each step replicating 1, 25, 100, 250, 500, or 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 Figure 5.1-a. 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 Figure 5.1-a, we can conclude that: i) the steps observed are due to the creation and transfer of several replicas along with the corresponding proxy-in/proxy-out pairs; ii) 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; iii) the incremental replication of 25 to 100 objects at each time is the most efficient alternative; iv) 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-



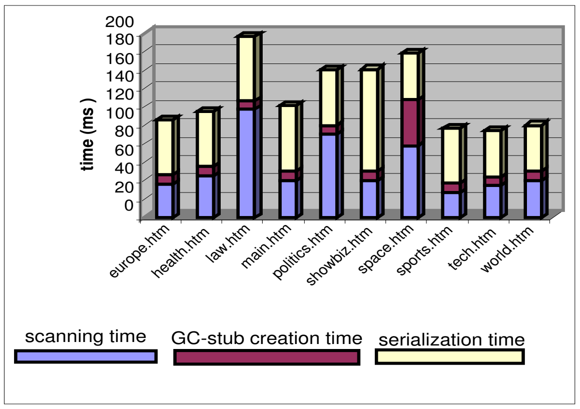
a) Incremental replication without clustering.



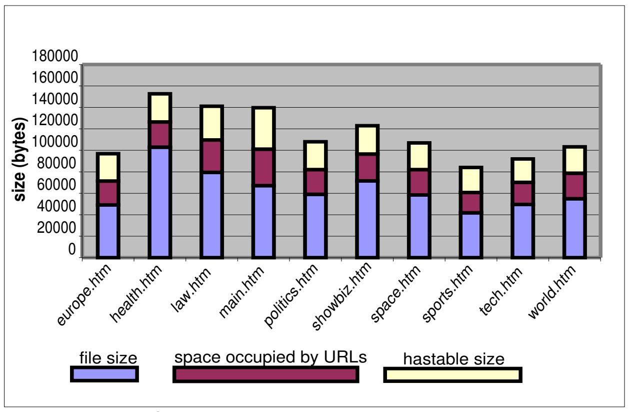
b) Incremental replication with clustering.

file size	number of URLs	scan time	GC-stub creation time	hash-table size	time to serialize
43563	326	38	3	19252	67

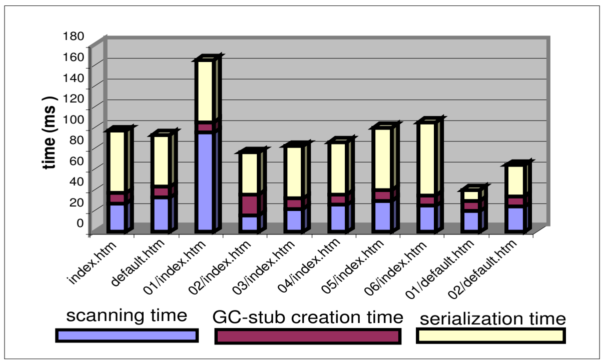
c) Mean values obtained with the 155 HTML files automatically downloaded from the cnn.com site (sizes in bytes and times in milliseconds).



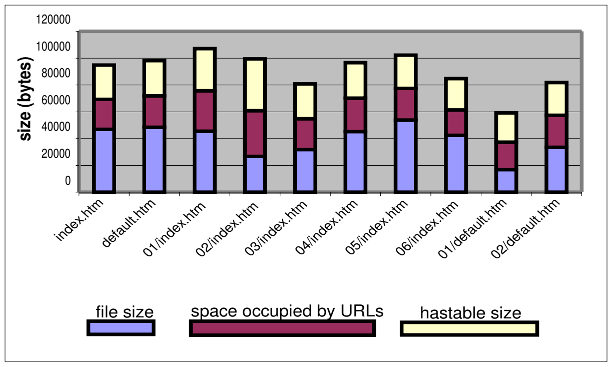
d) Time spent for the top group.



f) Space spent for the top group.



e) Time spent for the branch group.



g) Space occupied for the branch group.

Figure 5.1: Performance results for replication and distributed garbage collection.

in pairs; v) 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 the same of the previous section. 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-in/proxy-out pair being created. Consequently, each object can not be individually updated in P2.

The results are presented in Figure 5.1-b. 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 pairs. We can conclude that, w.r.t. the results without clustering, these results are: i) much better because there is only one proxy-out/proxy-in pair being created and transferred for each cluster; we observed that the most significant performance cost is due to data serialization (done by the JVM) and network communication; ii) 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 (i).

5.2 Distributed Garbage Collector

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 in (Sanchez et al. 2001).) 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 I and II. 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;[1] 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

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

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.

In Figure 5.1-c we present, for the 155 files: the mean file size, the mean number of URLs enclosed in each file, the mean time to scan a file, the mean time it takes to create a GC-stub in the corresponding hash table, the mean size of the hash table containing all the GC-stubs corresponding to all the URLs enclosed in a file (that depends on the size of the corresponding URL), and the mean time it takes to serialize a hash table with all the GC-stubs corresponding to a single file.

However, in a real situation, we expect that only some objects do 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 Figures 5.1-d and 5.1-e, we present, for each file of the two considered sets, the time spent in each relevant operation . In Figures 5.1-f and 5.1-g, 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.

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.

6

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, abusive resource consumption by mobile agents, and distributed garbage collection are automatically handled by the system.

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 in run-time how objects are invoked: RMI or local invocation on a replica.

Replicas are transparently created and mapped into processes; the programmer can control, in run-time, the amount of objects being replicated by creating dynamic clusters thus improving the performance of the system.

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

Finally, we showed performance results that are very encouraging.

6.1 Future Work

There are two immediate lines of work that can be pursued based on the work already done and the results achieved: i) make the distributed garbage collector complete, i.e., able to reclaim cycles of unreachable objects and ii) determine and evaluate the necessary (though few) modifications to the architecture to adapt it to compact devices and develop its implementation on these same devices.

The last subject to be addressed will be the development of memory and resource policies and profiles adaptable (adapted as an initial approach) to different execution environments. These issues that cannot be feasibly addressed without completion of at least part of the two prior aspects.

6.1.1 Complete DGC with Replicas

The collection of distributed cycles of garbage has been a field of active study for many years and presently, as well(Abdullahi and Ringwood 1998; Fessant 2001; Fessant et al. 1998; Louboutin and Cahill 1997; Rodrigues and Jones 1998; Shapiro et al. 2000). However, these solutions are not correct in the presence of replicated data. We intend to study solutions that are not disruptive to applications, that are scalable to large number of nodes, and correct in the presence of replicas, following our previous work(Sanchez et al. 2001; Veiga and Ferreira 2003). Since these approaches are not complete in the presence of replicas, i.e., they are not able to reclaim distributed cycles of replicated data, this is a very interesting subject for future work, studying possible combinations of these properties .

6.1.2 M-OBIWAN

We are currently adapting OBIWAN architecture to compact devices. This poses different challenges, both at research and technological levels. In technological terms there are several limitations in virtual machines adaptation to compact devices (MS .Net Compact Framework and Java 2 Micro Edition). For example, .Net CF does not provide programmatic reflection, remoting services nor explicit serialization. These are severe limitations that omit some of the mechanisms OBIWAN is based on. We are addressing these challenges by making OBIWAN compiler customizable and more sophisticated and developing a number of base runtime services which will be included in M-OBIWAN library.

In research terms, the sheer difference between desktop and compact devices capabilities raises a number of problems. Most related with this thesis work are those addressing memory management: in a compact environment, incremental replication uses bandwidth and memory more efficiently but applications cannot replicate objects indefinitely due to drastic memory limitations in these devices. This draws a new issue related with the third item in the initial section: memory policies. These include object de-localization[1], i.e., the ability to alleviate memory needs even of reachable objects, replacing them with proxies similar to those used to replicate the objects initially to the device. This way, some branches of the application object-memory graph are temporarily pruned. This reduces memory usage and preserves future access to swapped/dropped objects.

[1]object-dropping or swapping

7

Abdullahi, S. E. and G. A. Ringwood (1998). Garbage collecting the internet: a survey of distributed garbage collection. *ACM Computing Surveys (CSUR) 30*(3), 330–373.

Arnold, K. and J. Gosling (1996). *The Java Programming Language*. Addison-Wesley.

Baduel, L., F. Baude, and D. Caromel (2002). Efficient, flexible, and type group communication in java. In *Proc. of ACM Joint ACM Java Grande - ISCOPE 2002 Conference (JGI'02)*.

Butterwoth, P., A. Otis, and J. Stein (1991, October). The GemStone object database management system. *Communications of the ACM 34*(10), 64–77.

Caromel, D. (1993). Towards a method of object-oriented concurrent programming. *Communications of the ACM 36-99*, 90–102.

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

Demers, A. J., K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch (1994, 8-9). The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, Santa Cruz, California, pp. 2–7.

Deux, O. et al. (1991, October). The O_2 system. *Communications of the ACM 34*(10), 34–48.

Dias, P., C. Ribeiro, and P. Ferreira (2003, jun). Enforcing history-based security policies in mobile agent systems. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*.

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

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

Ferreira, P. and M. Shapiro (1996, may). Larchant: Persistence by reachability in dist. shared memory through garbage collection. In *Proc. 16th Int. Conf. on Dist. Comp. Syst. (ICDCS)*, Hong Kong. http://www-sor.inria.fr/SOR/docs/LPRDSMGC:icdcs96.html.

Ferreira, P. and M. Shapiro (1998, July). Modelling a dist. 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).

Fessant, F. L. (2001). Detecting distributed cycles of garbage in large-scale systems.

Fessant, F. L., I. Piumarta, and M. Shapiro (1998). An implementation for complete asynchronous distributed garbage collection. *ACM SIGPLAN Notices 33*(5), 152–161.

Franklin, M. J., M. J. Carey, and M. Livny (1997). Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems 22(3)*, 315–363.

General Magic, I. (1997). Introduction to the odyssey api. http://www.genmagic.com/.

Gharachorloo, K., S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill (1992, August). Programming for different memory consistency models. *Journal of Parallel and Distributed Computing 15*(4), 399–407.

Guerraoui, R. and A. Schiper (1997). Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS-17)*, Baltimore, USA, pp. 578–585.

Hagimont, D. and F. Boyer (2001, January). A configurable rmi mechanism for sharing dist. java objects. *IEEE Internet Computing 5*.

Hosking, A. L. and J. E. B. Moss (1993, September). object fault handling for persistent programming languages: a perfromance evaluation. In *ACM Conf. on Object-Oriented PRogramming Systems, Languages and Applications, 288-303*.

Keleher, P., A. L. Cox, and W. Zwaenepoel (1992, May). Lazy release consistency for software distributed shared memory. In *Proc. 19th Int. Symposium on Comp. Architecture*, Gold Coast (Australia), pp. 13–21.

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

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

Liskov, B., M. Day, and L. Shrira (1992, August). Distributed object management in Thor. In *Proc. Int. Workshop on Distributed Object Management*, Edmonton (Canada), pp. 1–15.

Louboutin, S. R. Y. and V. Cahill (1997). Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events. In *Proceedings of ICDCS'97 International Conference on Distributed Computing Systems*. IEEE Press.

Maheshwari, U. and B. Liskov (1995). Collecting cyclic dist. garbage by controlled migration. In *Proc. of PODC'95 Principles of Dist. Computing*. Later appeared in Dist. Computing, Springer Verlag, 1996.

ObjectSpace, I. (1997, September). Objectspace voyager core technology. Objectspace technical report, ObjectSpace, Inc.

Piquer, J. M. (1991, June). Indirect reference-counting, a distributed garbage collection algorithm. In *PARLE'91—Parallel Architectures and Languages Europe*, Volume 505 of *Lecture Notes in Computer Science*, Eindhoven (the Netherlands), pp. 150–165. Springer-Verlag.

Plainfossé, D. and M. Shapiro (1995, September). A survey of dist. garbage collection techniques. In *Proc. Int. W'shop on Memory Management*, Kinross Scotland (UK). http://www-sor.inria.fr/SOR/docs/SDGC_iwmm95.html.

Platt, D. S. (2001). *Introducing the Microsoft.NET Platform*. Microsoft Press.

Preguiça, N., C. Baquero, F. Moura, J. L. Martins, R. Oliveira, H. J. L. Domingos, J. O. Pereira, and S. Duarte (2000). Mobile transaction management in mobisnap. In *ADBIS-DASFAA*, pp. 379–386.

Preguiça, N., J. L. Martins, M. Cunha, and H. Domingos (2003). Reservations for conflict avoidance in a mobile database system. In *Proc. of the 1st Usenix Int'l Conference on Mobile Systems, Applications and Services (Mobisys 2003)*.

Preguiça, N., M. Shapiro, and C. Matheson (2002, May). Efficient semantics-aware reconciliation for optimistic write sharing. Technical report, Microsoft Research MSR-TR-2002-52.

Ribeiro, C., A. Zúquete, and P. Ferreira (2001, November). Enforcing obligation with security monitors. In *The Third International Conference on Information and Communication Security (ICICS'2001)*, Xi'an, China. Springer Verlag.

Ribeiro, C., A. Zúquete, P. Ferreira, and P. Guedes (2001, February). Spl: An access control language for security policies with complex constraints. In *Network and Distributed System Security Symposium (NDSS'01)*, San Diego, California.

Rodrigues, H. and R. Jones (1998). Cyclic distributed garbage collection with group merger. *Lecture Notes in Computer Science 1445*.

Rodriguez-Rivera, G. and V. Russo (1997). Cyclic distributed garbage collection without global synchronization in corba. In *OOPSLA97-gc*.

Sanchez, A., L. Veiga, and P. Ferreira (2001, January). Dist. garbage collection for wide area replicated memory. In *Proc. of the Sixth USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio (USA).

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

Shapiro, M., P. Dickman, and D. Plainfossé (1992, August). Robust, distributed references and acyclic garbage collection. In *Symposium on Principles of Distributed Computing*, Vancouver, Canada, pp. 135–146. ACM Press.

Shapiro, M., F. L. Fessant, and P. Ferreira (2000). Recent advances in distributed garbage collection. *Lecture Notes in Computer Science 1752*, 104.

Silva A., Romão A., D. D. M. d. S. M. (2001). Towards a reference model for surveying mobile agent systems. In *Autonomous Agents and Multi-Agents Systems, 4, 187-231.*

Skubiszewski, M. and N. Porteix (1996, April). GC-consistent cuts of databases. Rapport de recherche 2681, Institut National de Recherche en Informatique et Automatique, rocquencourt.

Sousa, P., M. Sequeira, A. Zúquete, P. Ferreira, C. Lopes, J. Pereira, P. Guedes, and J. Marques (1993, nov). Distribution and persistence in the ik platform. *Computing Systems Journal 6(4).*

Veiga, L. and P. Ferreira (2001, Nov). Mobility and wireless support in OBIWAN. In *3rd IFIP/ACM Middleware Conference*, Heidelberg (Germany).

Veiga, L. and P. Ferreira (2002a, July). Incremental replication for mobility support in OBIWAN. In *The 22nd International Conference on Distributed Computer Systems*, Viena (Austria), pp. 249–256.

Veiga, L. and P. Ferreira (2002b, Sep). Mobility support in OBIWAN. In *2nd Microsoft Research Summer Workshop*, Cambridge (UK).

Veiga, L. and P. Ferreira (2003, Mar). Repweb: Replicated web with referential integrity. In *18th ACM Symposium on Applied Computing (SAC'03)*, Melbourne, Florida, USA.

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

Wilson, P. R. and S. V. Kakkad (1992, September). Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Int'l W'shop On Object Orientationin Operating Systems. Paris, France, 364-377*.

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

A

Planeamento Inicial

Durante a fase inicial do Doutoramento, proponho-me fazer uma investigação bibliográfica da área da computação móvel relacionada com os modelos de dados e de construção de aplicações. Esta actividade passará pela familiarização com as principais conferências e grupos de investigação deste tema e com os seus resultados, quer publicações, quer protótipos informáticos. Desta investigação pretende-se que resultem uma melhor compreensão das questões científicas que se colocam neste domínio, um documento que constituirá o núcleo do capítulo de trabalho relacionado da dissertação e um documento onde estejam propostos os objectivos específicos da investigação a realizar.

Uma vez completa esta fase, pretendo prosseguir os trabalhos de investigação e concretização das soluções encontradas. Esta tarefa será a mais susceptível de um prolongamento em função da rapidez de obtenção de resultados e da quantidade de trabalho docente a realizar paralelamente.

Finda esta fase, pretendo continuar a avaliação dos protótipos desenvolvidos com vista à sua optimização e produção de resultados mais detalhados visando a publicação do trabalho desenvolvido. Durante o período de escrita de dissertação será redigida a dissertação tomando como base a proposta de tese apresentada anteriormente e os resultados das publicações realizadas no âmbito do tema de Doutoramento. O planeamento desta tese de doutoramento encontra-se resumido nas alíneas seguintes:

a) Estudo do Trabalho Relacionado e Planeamento de Investigação

Duração estimada : 12 meses

Data de Início: Março de 2002

Data de Conclusão: Final de Março de 2003

b) Investigação e Concretização de Soluções

Duração estimada: 24 meses

Data de Início: Abril de 2003

Data de Conclusão: Março de 2005

c) Avaliação e Optimização do(s) Protótipo(s) Desenvolvido(s)

Duração estimada: 6 meses

Data de Início: Março de 2005

Data de Conclusão: Setembro de 2005

d) Escrita da Dissertação

Duração estimada: 6 meses

Data de Início: Setembro de 2005

Data de Conclusão: Março de 2006

e) Provas

Data de Conclusão: após Março de 2006

Produção Intelectual Esperada

Este doutoramento pretende desenvolver contribuições na área da Gestão de Memória em Computação Ubíqua, focando nomeadamente os seguintes aspectos:

- Replicação incremental de grafos de objectos e tratamento, de forma transparente, de faltas de objectos em sistemas distribuídos e integrados em redes ad-hoc.

- Reciclagem automática de memória distribuída com e sem replicação em sistemas distribuídos.
- Clustering dinâmico de objectos.
- Modelação de gestão de memória, em sentido mais vasto, em redes ad-hoc de dispositivos com recursos limitados, prevendo a deslocalização de dados ainda activos.

Enquadramento

A cada vez maior implantação de dispositivos móveis de computação, que não apenas os computadores tradicionais, tem conduzido a um relaxamento das formas de organização das redes informáticas. Enquanto que as redes constituídas por computadores estacionários têm topologias estáticas que permitem a introdução de forma simples de mecanismos de configuração, administração e partilha de dados, as redes constituídas por dispositivos móveis de computação surgem com uma índole mais dinâmica.

Assim, da perspectiva dos utilizadores, esta realidade nova introduz uma maior flexibilidade geográfica e funcional na utilização dos computadores. Contudo, na vertente da concepção de mecanismos de suporte às aplicações (sistemas operativos, middleware), apresentam-se com maior exigência, neste novo ambiente variável, questões de adequação, coerência e adaptabilidade no que concerne o ambiente de execução de aplicações.

Este tipo de problemas enquadram-se na área de investigação actualmente denominada computação ubíqua e que engloba ainda áreas como as infra-estruturas de rede móveis, a gestão de recursos em dispositivos móveis, protocolos para redes ad-hoc, os interfaces pessoa-máquina, redes pessoais e dispositivos incorporados no utilizador (wearable computing), etc

Os pontos anteriores podem agrupar-se em três classes: suporte à execução, reciclagem de memória clássica e gestão de memória num sentido mais vasto.

O primeiro ponto visa o suporte à realização e adaptação de aplicações distribuídas aos novos ambientes de computação mencionados anteriormente. Estes podem ser compostos por, além dos computadores tradicionais, também por dispositivos muito diferentes e de recursos - processador, memória, rede, memória persistente - limitados. Assim, a replicação incremental de grafos de objectos e o tratamento de faltas de objectos em sistemas distribuídos integrando redes ad-hoc, constituem um suporte fundamental para que a computação distribuída, tal como existe hoje, possa ser correctamente estendida de forma a poder, simultaneamente, comportar aplicações sofisticadas e adaptar-se a dispositivos de recursos limitados.

O segundo ponto consiste no estudo da reciclagem automática de memória distribuída, com e sem replicação, realizada sobre sistemas distribuídos integrando redes ad-hoc. A estes algoritmos, na sua correcção e funcionamento, impõem-se restrições não exigidas aos algoritmos actualmente existentes. Importa portanto, estudar de que forma os algoritmos actuais podem ser adaptados, ou combinados, para satisfazer estas novas exigências ou, tal não sendo possível, enveredar pela concepção de outros que cumpram o especificado. As soluções adoptadas deverão respeitar, não apenas os tradicionais critérios de correcção, completude e diligência, mas também imperativos de escalabilidade, assincronismo e latência reduzida.

Um prossecução natural deste ponto será o esforço de abordagem das questões levantadas a um nível de abstracção mais elevado e que permita a configuração do suporte de execução a diferentes dispositivos e a sempre diferentes configurações de recursos numa rede ad-hoc. Os dois últimos pontos enquadram-se, portanto, numa noção mais vasta e lata da gestão de memória em computação ubíqua e pretende, firmando-se nos pontos anteriores, desenvolver modelos de gestão de memória que permitam optimizar, adaptando-o às novas realidades, tanto a replicação de objectos como a reciclagem dos grafos distribuídos por eles criados. Incluem-se assim, necessariamente, aspectos como o clustering dinâmico de objectos, a deslocalização de dados e, naturalmente, o desenho de políticas de gestão de memória.

Marcos De Realização Alcançados (Maio 2003)

Neste momento já foi desenvolvida pesquisa no âmbito do trabalho relacionado respeitante às vertentes de replicação e de reciclagem automática de memória distribuída.

Foi, também, já iniciado trabalho de investigação no âmbito destas vertentes que foi apoiado na participação em algumas conferências e workshops, tendo sido atingida a concretização de soluções preliminares relevantes com as publicações conseguidas.

Publicações no decurso do trabalho com vista à obtenção do doutoramento (com *refereeing*)

- Mobility and Wireless Support in OBIWAN. Luís Veiga, Paulo Ferreira. In Advanced Topic Workshop on Middleware for Mobile Computing, **3rd IFIP/ACM Middleware Conference**, Heidelberg, Germany, Nov. 16, 2001.
- Incremental Replication for Mobility Support in OBIWAN. Luís Veiga, Paulo Ferreira. In **22nd IEEE International Conference on Distributed Computing Systems (ICDCS'02)**, Vienna, Austria, July 2-5, 2002.
- Mobility Support in OBIWAN. Luís Veiga, Paulo Ferreira. In **2nd Microsoft Research Summer Workshop**, Cambridge, UK Sep. 9-11 2002.
- RepWeb: Replicated Web with Referential Integrity. Luís Veiga, Paulo Ferreira. In **18th ACM Symposium on Applied Computing (SAC'03)**, Melbourne, Florida, USA, Mar. 9-12, 2003.

Submissões já realizadas ainda em apreciação

- OBIWAN - Design and Implementation of a Middleware Platform. Paulo Ferreira, Luís Veiga, Carlos Ribeiro. Submetido a **IEEE Transactions on Parallel and Distributed Systems**.

Apresentações Convidadas

- Incremental Replication in OBIWAN with Microsoft .Net. Apresentação no **Microsoft Research Crash Course .Net** 2, 25-28 Março, Cambridge, 2002.

Participação, sob convite ou bolsa, em Conferências e Workshops sem publicação

- **5th Usenix Symposium on Operating Systems Design and Implementation (OSDI'02)**, Boston, 9-11 Dez, com bolsa atribuída pela Usenix.
- **Microsoft Mobility Developer's Conference**, Abril, Londres, 2002.
- **1st Microsoft Research Rotor Project Workshop**, 22-27 Julho, Cambridge, 2002.