Seamless Service Access via Resource Replication

Paulo Ferreira Luís Veiga

INESC-ID/IST, Rua Alves Redol N^o 9, 1000-029 Lisboa, Portugal

{paulo.ferreira, luis.veiga}@inesc-id.pt

Abstract

Replication is a well-known technique for improving data availability and application performance as it allows to collocate data and code. However, there are several relevant difficulties that must be solved to take full advantage of replication; we address the following: i) replica management, ii) memory management, and iii) adaptability.

We present an archetypical architecture for mobile middleware that is used along this chapter, the mechanisms supporting how and which data is replicated (both for the object and file models), the solution for the garbage collection of replicas, and the policies allowing applications to control objects replication.

Contents

1 Introduction

Replication is a well-known technique for improving data availability and application performance as it allows to collocate data and code. Thus, data availability is ensured because, even in presence of network failures, data remains locally available; in addition, application performance is potentially better (when compared to a remote invocation approach) as all accesses to data are local.

However, there are several relevant difficulties that must be solved to take full advantage of replication. In this chapter we address the following: i) replica management, ii) memory management, and iii) adaptability. Note that, many other issues are equally important [1],

e.g., how to merge or reconcile replicas that, due to updates performed independently, have diverged; however, such issues are considered elsewhere in this book.

Replica management is related to the fundamental issues of knowing which and how data should be replicated. Memory management addresses the need to ensure that the memory of mobile devices (PDAs, laptops, etc.) is occupied with useful data. This includes two main tasks: i) freeing the memory occupied by useless replicas, which can be done by garbage collecting such replicas, and ii) swapping out useful data to disk or to remote computers. In addition, for object based applications, memory management is responsible for ensuring referential integrity of the objects graph. Adaptability is the capability applications have to control and adapt to the resources they use (memory, network, etc.) in order to better deal with the variability of mobile environments; such variability affects network bandwidth, network connection or disconnection, amount of available memory, etc.

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

The fact that mobile devices impose severe constraints in terms of such resources raises the importance of the following: i) the data to be replicated should be the one that is really needed, so that memory is not wasted, ii) replicated data that is no longer needed must be automatically detected and garbage collected, thus releasing the memory occupied, and iii) the underlying middleware must support flexible mechanisms so that applications can react and adapt to the dynamics of mobile environments (e.g., variable network availability, amount of free memory on the device, etc.).

Finally, portability and programmability are also relevant aspects that must be taken into

account by the mobile middleware. As a matter of fact, mobile environments are characterized by the heterogeneity of devices, operating systems, virtual machines, etc. So, the mobile middleware should be, as much as possible, independent from such differences in order to be portable to a wide range of platforms.

Programmability means that the middleware should release applications programmers from dealing with system level issues while providing a familiar application programming interface (API). Thus, the mobile middleware should not imply the modification of neither the operating systems nor the virtual machines, and should not impose radically new APIs.

In the remainder of this chapter we first clarify the programming model being considered. Then, we present the archetypical architecture for mobile middleware that is used along this chapter, the mechanisms supporting how and which data is replicated (both for the object and file models), the algorithms for the garbage collection (GC) of replicas, and the policies allowing applications to control objects replication.

2 Programming Models

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

Concerning the issues addressed in this chapter, the relevant characteristic of such paradigms is their ability, or lack of it, to support arbitrary graphs of data. As a matter of fact, as explained afterwards, the existence of data graphs has a strong impact when deciding which and how data must be replicated.

The object-oriented paradigm naturally supports such notion of data graphs; the same applies to structured files whose contents include references to other files (e.g., graphs of HTML

files connected by URLs). In this chapter we consider both cases: i) applications may use arbitrary graphs of data, and ii) applications simply use plain unstructured data (i.e., without containing references that allow to build data graphs). In the first case we use the term object to designate a *datum* that can be an instance of a class, an HTML file, etc. Thus, an object is simply a set of bytes that may contain references to other objects. In the second case, we use the term file to designate a *datum* that is a set of unstructured bytes (thus, without the notion of reference).

Given that the programming model in which applications handle arbitrary data graphs is the most widely used and is highly flexible, in this chapter we focus our attention on this case; we call it the object model. We also consider the file model because file system support is widespread and is well known both by users and applications programmers. In this model the mobile middleware offers a file-based API (possibly extended with replication specific functions) in which there are no references between files.

The object model is arguably the most widely adopted programming model. Naturally, object replication has been addressed in several projects such as Thor [2, 3], OceanStore [4, 5, 6], DENO [7], OBIWAN [8, 9], M-OBIWAN [10, 11], DERMI [12], Javanaise [13, 14], GoldRush [15], Alice [16], replicated CORBA [17], among others. However, not all have addressed with the same level of concern the challenges raised by mobility environments.

Relevant projects supporting file replication include CODA [18, 19], which was the first to address the issue of disconnected work in distributed file systems; Ficus [20], Rumor [21], and Roam [22, 23] represent a line of very interesting work concerning distributed file systems with growing concerns of mobility, leading to the concept of selective replication, i.e., a mechanism by which only the files that applications really need to access are replicated, instead of a whole volume (more details in Section 4.2).

We do not consider the programming paradigm based on relational databases given that applications developed according to this paradigm are mostly query-oriented, instead of navigation-based (on a graph). Operations on data are declaratively defined by SQL-queries for insertion,

update and removal of records. Data on different tables can be joined by matching field values. Several queries can be composed into transactions guaranteeing ACID properties [24]. In mobile computing, these ACID properties are relaxed in order to provide acceptable consistency requirements while allowing concurrent update on replicas placed in different nodes. One influential work regarding database replication is Bayou [25, 26, 27] in which the merging of replicas is specially considered. Replication of relational databases in mobile computing has also been addressed in Mobisnap [28].

3 Architecture

Replication can be used both in client-server (CS) [29] or peer-to-peer (P2P) [30] distributed systems. In CS systems, data is replicated from a server into the mobile client device, where it is accessed; then, if needed, data is sent back to the server. The server (or servers, as many may exist) have the fundamental purpose of storing data persistently. In a CS architecture, data is shared among mobile clients always by intermediation of a server.

With a P2P architecture, any computer 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 running on a mobile device can either request the local creation of replicas of remote data (P acting as a client) or be asked by another process to provide data to be replicated (P acting as a server). Hybrid architectures consider the coexistence of both approaches.

The P2P approach is generally more flexible than a CS architecture given that mobile devices are free to replicate data among them (for example, epidemically [31]); this raises delicate aspects in terms of consistency that are considered elsewhere in this book. For CS and P2P architectures, the issues of replica management, memory management, and adaptability are equally relevant.

For clarity, in the remaining sections of this chapter, when addressing these issues for the object model, we refer to Figure 1; it presents an archetypal architecture illustrating the most

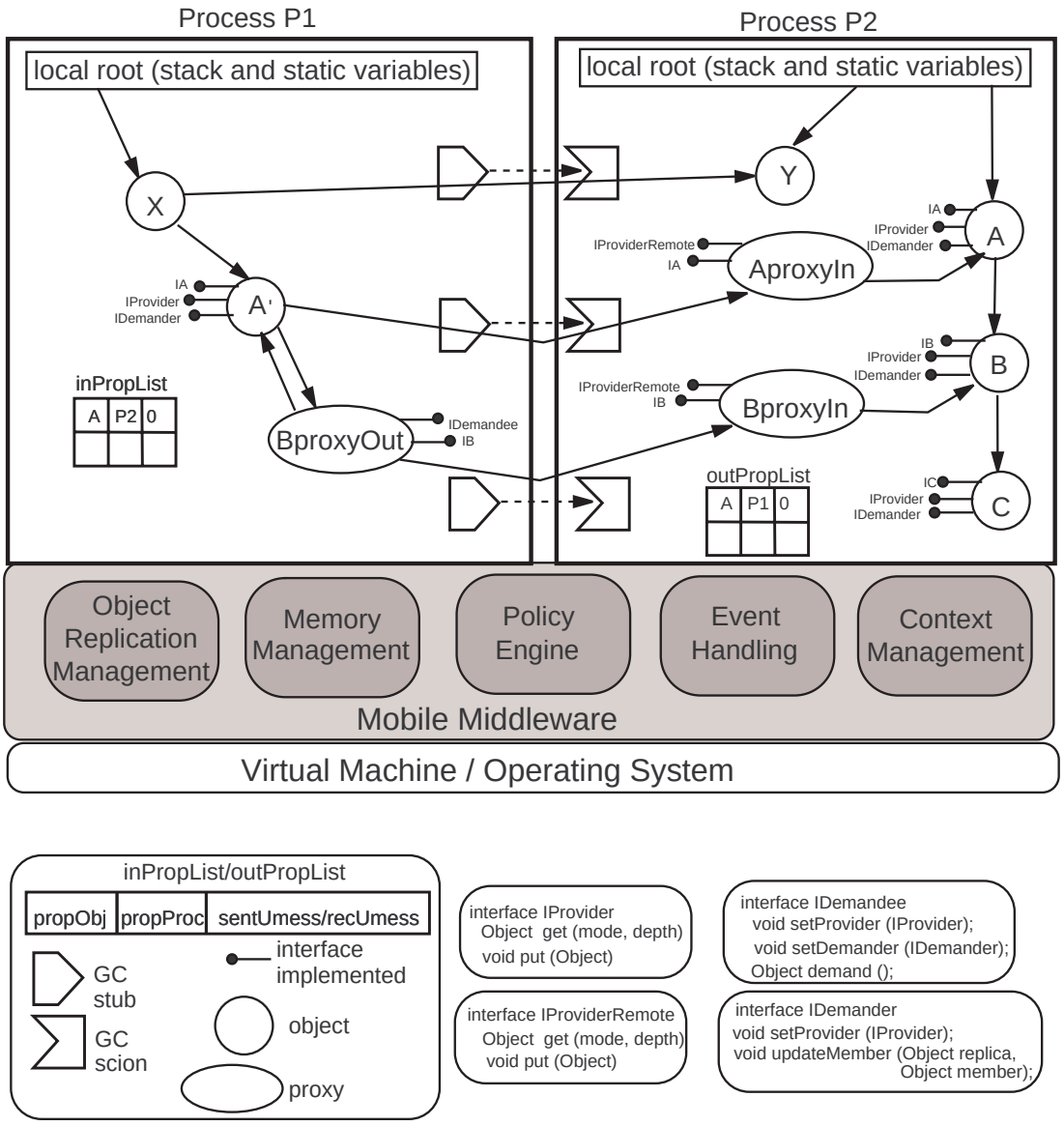

Figure 1: Archetypical architecture for mobile middleware supporting object replication.

important data structures concerning the replication of an objects graph.

Objects X, Y, A, B and C are created by the application; their replicas (A', B', etc.) are created either upon the programmer's request or automatically (i.e., by the middleware, without having been explicitly required by the application code, but resulting from its execution). Proxies-in and proxies-out, as well as references pointing to them, are part of the middleware and are transparent to the programmer.

Without loss of generality, we assume that processes P1 and P2 run in two different computers, and the initial situation is the following: i) P2 holds a graph of objects Y, 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 object A does not distinguish B' from B as they offer the same interface; and v) object X points to Y.

The most relevant data structures are the following:

- **Proxy-out/proxy-in pairs** [32]. 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 4.1 we describe how these proxies help in supporting object replication.

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

  We are assuming that GC-stubs and GC-scions do not impose any indirection on the native reference mechanism. In other words, they interfere neither with the structure of references nor with the invocation mechanism. They are simply GC specific auxiliary data structures. Thus, GC-stubs and GC-scions should not be confused with (virtual machine) native stubs and scions (or skeletons) used for remote method invocations (RMI).

- **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 distributed GC purposes) has been sent/received (more details in Section 5).

Another important aspect of the archetypal architecture is the functionality supported by the above mentioned data structures as well as the interfaces they implement. In particular, proxies-out, being one of the main entities of the mobile middleware responsible for the replication mechanism, have to implement the same interface of the object they are replacing. In addition, for reasons that will be made clear in Section 4.1, application objects also have to implement a few methods that are, in fact, middleware code. However, these methods are not written by the application programmer; they are automatically generated either at compilation or at runtime.

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 invoked on these objects. (The same reasoning applies to objects X and Y; however, given that they are not involved in the replication scenario, we do not consider them.)

- **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 another process (possibly 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 objects graph (as described in Section 4.1).

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

In addition to the data structures already presented, there are five other modules of the mobile middleware (addressed in the following sections): i) object replication management provides the mechanisms supporting data replication, ii) the memory management module is responsible for the distributed garbage collection of replicas, iii) the policy engine module triggers or mediates responses to events occurred in the system, iv) the event handling module registers the relevant events occurred in the system, and v) the context management module abstracts resources and manages the corresponding properties whose values vary during applications execution.

When considering a file model, i.e., a mobile middleware that provides a (possibly extended) well known traditional file-based API, the archetypal architecture does not contain some of the data structures mentioned, for obvious reasons. In particular, the GC-stubs/GC-scions are not needed. However, Figure 1 is still valid; for example, the lists InPropList/OutPropList are still needed so that the system keeps track of which files were replicated into/from, the policy engine still allows the application to deal with mobile environment dynamics, etc.

4 Replica Management

A fundamental issue concerning data replication is the impact of this mechanism on the API. In other words, it is crucial that applications programmers are not forced to deal with system details concerning which and how data is effectively replicated. Such system issues (e.g., object faulting and resolving) must be handled transparently by the underlying middleware while data gets replicated from one computer to another. For example, when considering the object model, in which distributed applications access a data graph, the referential integrity of the graph must be ensured in order to avoid dangling references.

Therefore, in this section we address the following issues: i) the mechanisms supporting

objects and files replication, and ii) how objects and files are chosen to be replicated. These issues are analyzed taking into account the fact that mobile devices impose strong constraints in terms of available memory and network bandwidth.

4.1 How to Replicate

Object replication differs significantly from file replication. The difference results from the fact that, with the object model, applications access data by navigating on objects graphs. Such navigation does not occur when applications access plain unstructured files. This difference has an important impact on deciding how and which data should be replicated.

When an application running on a mobile device navigates on an objects graph, its execution proceeds normally as long as all objects are local. However, when the target object is not yet locally replicated, this generates an object fault that must be resolved by the middleware. Therefore, the above mentioned aspects concerning the replication of objects (how and which) are strongly dependent on the navigation performed by the application.

Regarding the file model, in which files are plain unstructured streams of bytes, applications do not navigate on a graph while executing (in opposition to the case in which the object model is used). Thus, applications access local files by means of the traditional system calls "open", "read", "write", etc. offered by most operating systems, possibly extended by the mobile middleware with replication specific capabilities. Among other functionalities, these extensions improve the memory usage in mobile devices by compressing file contents or replicating just the blocks that are really needed, for example. An interesting situation in which files are replicated, possibly resulting from an explicit request from the user, is the hoarding of files from a desktop computer into a laptop or PDA (more details in Section 4.2).

4.1.1 Object Model

Given the memory restrictions imposed by mobile devices (when compared to desktop computers) the replication of objects can not be done simply by serializing all the object graph

in the originating computer and send it to another one. Such an approach (which is available when using Java [33] or .Net [34] platforms) is clearly inappropriate due to the large amount of computing, communication and memory resources required. Thus, the mobile middleware must handle incremental replication, i.e., the partial replication of the objects graph.

The incremental replication of an objects graph has two clear advantages when compared to the replication of the whole reachability graph in one step: i) the latency imposed on the application is lower because it can immediately invoke the new replica without waiting for the whole graph to be available, and ii) only those objects that are really needed become replicated, thus reducing the memory and network bandwidth required.

Therefore, the situations in which an application does not need to invoke every object of a graph, or the computer where the application is running has limited memory and/or network bandwidth available, are those in which incremental replication is most useful. However, there are situations in which it may be better to replicate the whole graph; for example, if all objects are really required for the application to work, if there is enough memory, and the network connection will not be available in the future, it is better to replicate the transitive closure of the graph. The mobile middleware must allow the application to easily make this decision at runtime, i.e., between incremental or transitive closure replication mode.

Taking into account the archetypal architecture presented in Figure 1, we now describe how objects can be incrementally replicated from a process P2 into another process P1. (Note that, in the initial situation, A' was replicated the same way that B will be, as explained afterwards.) 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 detect and resolve the corresponding object fault. All IB methods in BProxyOut simply invoke on itself a special method (which is part of the mobile middleware) called BProxyOut.demand (belonging to interface IDemandee); this method runs as follows (see Figure 2 in which the numbers refer to the enumerated items below):

1. invokes (remotely) method BProxyIn.get in P2;

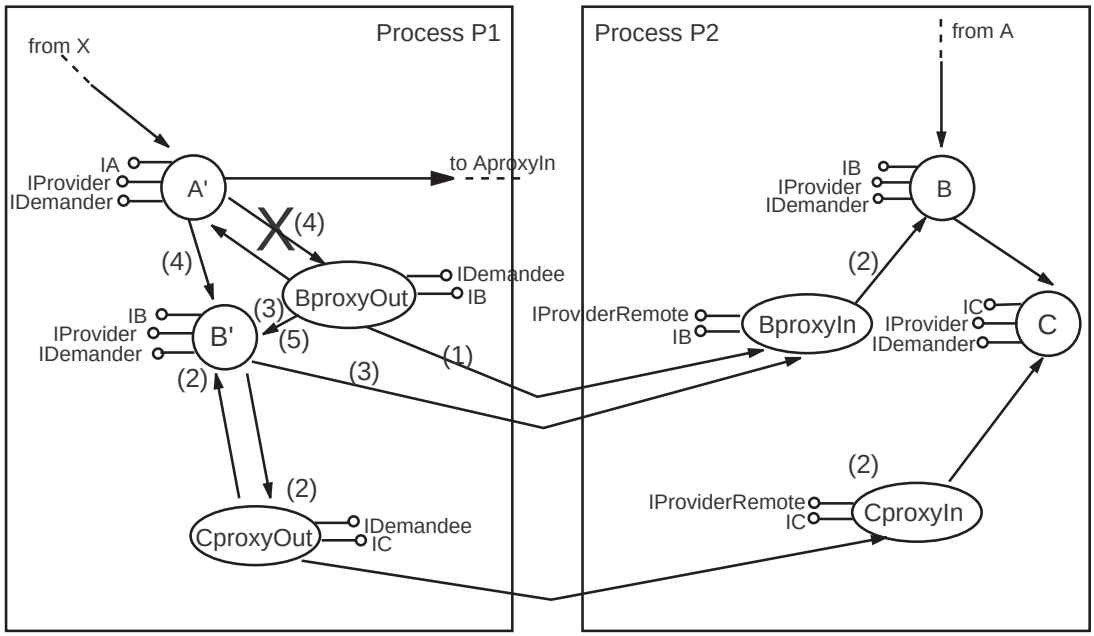

Figure 2: Incremental object replication.

2. BProxyIn.get invokes method B.get, that proceeds as follows: creates B', CProxyOut, CProxyIn and sets the references between them; once this method terminates, B', BProxyOut and CProxyOut are serialized and sent to P1 (CProxyIn remains in P2); note that A' still points to BProxyOut (and vice-versa);

3. BProxyOut invokes B'.setProvider(this.provider) so that B' also points to BProxyIn; this is needed because the application can later decide to update 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 worthy 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, (if and) 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 resolved 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 the portability of the mobile middleware supporting incremental replication.

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

A cluster is a set of reachable 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 middleware must allow the amount of objects in the cluster to be determined at runtime by the application. The application specifies the depth of the partial reachability graph that it wants to replicate as a whole. This is an intermediate solution between incrementally replicate each individual object or replicating the whole graph.

4.1.2 File Model

Concerning file replication, the previously mentioned memory and network bandwidth restrictions of mobile devices are obviously equally valid. Thus, the mobile middleware, possibly with cooperation from the operating system, must minimize both the space occupied by replicated files and network usage. This has been done in previous distributed file systems for fixed networks, either by compressing file contents or by replicating just the needed blocks (instead

of whole files) [35].

A very interesting and promising approach aimed at reducing the amount of memory consumed by replicated files explores replica contents similarities [36, 37, 38]. The idea consists of applying the SHA-1 hash function [39] to portions of each replica's contents; each portion is called a chunk. The probability of two distinct inputs to SHA-1 producing the same hash value is far lower than the probability of hardware bit errors. Relying on this fact, the obtained hash values can be used to univocally identify their corresponding chunk contents. From this assumption, if two chunks produce the same output upon application of the SHA-1 hash function, then they are considered to have the same contents. If both chunks are to be stored locally at the same computer, then only the contents of one of them needs to be effectively stored. Note that, in a similar way, if one of the chunks is to be sent to a remote machine that is holding the other chunk, the actual transference of the contents over the network can be avoided.

A content-based approach is employed to divide replica contents into a set of non-overlapping chunks, based on Rabin's fingerprints [40]. As a result, chunks may have variable sizes, depending on their contents. An important property of such chunk division approach is that it minimizes the consequences of insert-and-shift operations in the global chunk structure of a replica. The expected average chunk size may, however, be parameterized by controlling the number of low-order bits from the fingerprint's output that are considered by the chunk division algorithm. Moreover, to prevent cases where abnormally sized chunks might be identified, a minimum and maximum chunk dimension is imposed.

On each file system peer there is a common chunk repository which stores all data chunks, indexed by their hash value, that comprise the contents of the files that are locally replicated (see Figure 3). The data structures associated with the content of locally replicated files simply store pointers to chunks in the chunk repository. Hence, the contents of an update to a replica, or the whole replica, consists of a sequence of data chunks, stored in the chunk repository.

When a file replica is written, a data chunk is either created or modified. Then, its hash value is calculated and the chunk repository is examined to determine if an equally hashed

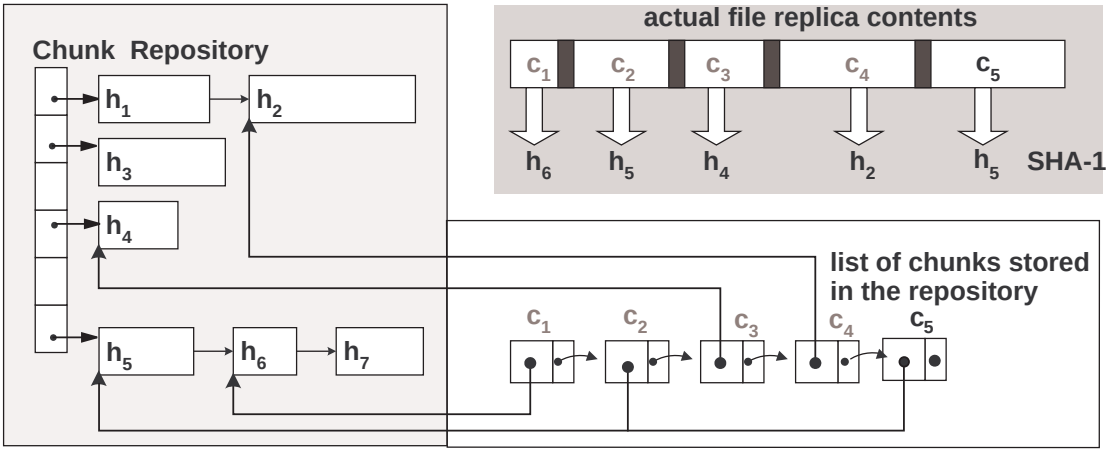

Figure 3: A file replica contents, the corresponding list of chunks, and the repository including them.

chunk is already stored. If not, a new entry corresponding to the new chunk is inserted in the repository. If a similar chunk already exists, a new pointer to that chunk is used. So, if different file replicas or versions of the same file replica contain data chunks with similar contents, they share pointers to the same entry in the chunk repository, thus reducing memory usage by the file system.

Replicating a file from one computer to another also makes use of the chunk repositories at each peer. When a chunk has to be sent across the network to another peer, only its hash value is firstly sent. The receiving peer then looks up its chunk repository to see if that chunk is already stored locally. If so, it avoids the transference of that chunk's content and simply stores a pointer to the already existing chunk. Otherwise, the chunk contents are sent and a new chunk is added to the repository.

To deal with the deletion of unused chunks from a repository, each chunk maintains a counter that is incremented each time a new pointer is set to that chunk. Conversely, that counter is decremented when a pointer to it is removed from the system's structures. This can occur when a previously replicated file is removed from the set of replicated files.

4.2 What to Replicate

Previous research done on replication has been mostly focused either on reducing the latency perceived by applications or as a means to increase the fault-tolerance of servers. However, in a mobile environment, replication main goal is to allow applications to keep on doing useful work in disconnected mode or when the network bandwidth is rather small; at the same time, the mobile middleware must optimize memory usage. Thus, deciding which data should be replicated is an important issue as replicating data that will not be needed by applications means that memory and network is being wasted.

4.2.1 Object Model

For the object model, in which applications access objects by navigating in the graph, replication arises naturally when resolving an object fault. In other words, when the application running in a mobile device invokes an object that is not yet locally mapped, it generates an object fault that is automatically resolved by the underlying middleware. The resolution of the object fault consists in requesting the faulted object from another computer as described in Section 4.1.1. This computer answers with the object faulted and possibly a few more that are anticipated to be needed shortly. This anticipation, i.e., the replication in advance of some objects, is based on the fact that objects are only accessed by means of graph navigation. So, the objects that must be replicated are those that can be accessed from the ones already replicated.

Then, an important aspect is the level of control and flexibility the middleware supports with respect to the amount of objects that are replicated in advance. For example, allowing an application to specify at runtime which branch of a graph and how many objects should be replicated. This aspect is addressed in Section 6.

The particular values of such options (branch and depth of the graph to be replicated) depends on many factors. Such a decision can be done manually by the programmer or automatically by the middleware (or both combined). In the first case the programmer may annotate his code with hints that the middleware will use accordingly; in the second case, the middleware

bases its decision on the past behavior of applications. In one hand, programmer annotations can be difficult to construct and are error prone; on the other hand, past behavior may not be available and it is not necessarily a good indicator of future application behavior. This issue is out of the scope of this chapter. However, an interesting possibility is to base the above mentioned values on context information [41].

4.2.2 File Model

Concerning the file model, the issue of deciding which files to replicate is equally important. However, given that there is no graph in which applications navigate, deciding which files to replicate must be done differently. In particular, the lack of a data graph means that there is no path the middleware can rely on to predict which files will be accessed in the future by an application.

Solutions to this replication problem (also known as hoarding) can be grouped as follows: i) solutions based on actions explicitly performed by the user stating which files should be replicated, or ii) solutions provided by the middleware that may take advantage of user provided hints.

The first case is concerned with a scenario in which, before disconnecting a PDA or a laptop from the network, a user explicitly replicates a set of files (from a desktop computer or a server), so that while disconnected he can still do his work. Note that this explicit replication can also be done using specific information provided by the user describing the files to be hoarded; based on this information, the system automatically replicates such files. For example, systems such as CODA [19] and SPY [42] require users to explicitly specify their hoard set. So, the user actions are decisive with respect to both aspects of file replication: grouping files together given that they are strongly related, and deciding which files (or groups of files) should be replicated given that they will certainly be used in the future. We will not consider this case further as it relies mostly on user explicit actions.

In the second case, the middleware has a much more active role; the following approaches

can be used.

- While working, the user provides the middleware with specific instants on time during which a specific task is being performed; the middleware, during each interval, detects which files are accessed and assumes that they are all strongly related and needed for the task under consideration.

- The middleware performs file replication based on the notion of semantic distance [43], with no help from the user. This notion relies on the temporal data usage patterns of file accesses; basically, those files that are accessed during a certain time interval are assumed to be strongly semantically related.[1] For this purpose, the middleware continuously monitors file accesses and clusters files accordingly.

Using file accesses to group related files is difficult because often it is not clear whether a sequence of file accesses is related or not. Furthermore, files that are related but are not accessed, at least for some time, may loose their privileged relation to other files.

Note that these two approaches mainly address the problem of finding which files are strongly related so that they should be replicated together. However, there is still the problem of predicting which files (or groups of files) will be used in the future (in particular, while the mobile device is completely disconnected or has severe network bandwidth limitations). Such forecasting can be done assuming that recently used files will certainly be accessed in the near future. This is, in fact, a LRU (Least Recently Used) approach that has been extensively used in the past for virtual memory support, and has been adopted by a hoarding system [44] with some refinements; e.g. by allowing the user to bound the time interval under consideration by the LRU. Such enhancement of the LRU approach requires user intervention; it basically consists of a user provided hoard profile that requires intensive user intervention. However, users should not have to worry about other issues than their work. Thus, a replication system should minimize user intervention and especially the amount of user attention required.

[1] The access patterns are in fact more sophisticated than this; e.g., spatial storage locality can be used as well.

5 Memory Management

In this section we address the issue of memory management: i) how the mobile middleware detects and deletes useless replicas, and ii) how to swap-out useful data to disk or to remote computers. This is a very relevant issue because memory is a scarce resource in mobile devices. In addition, when considering the object model in which applications navigate on a graph of objects, it is fundamental that the mobile middleware ensures the referential integrity of the objects graph. As described in this section, this is ensured by means of automatic memory management, also known as garbage collection (GC), which also detects and deletes useless replicas.

It is widely recognized that manual memory management (explicit allocation and freeing of memory by the programmer) is extremely error-prone leading to memory leaks and dangling references. Memory leaks consists in data that is unreachable to applications but still occupies memory, because its memory was not properly released. Dangling references are references to data whose memory has already been (erroneously) freed; later, if an application tries to access such data, following the reference to it, it fails. Such failure occurs because the data no longer exists or, even worse, the application accesses other data (that has replaced the one erroneously deleted) without knowing.

Memory leaks in servers and desktop computers are known to cause serious performance degradation. In addition, memory exhaustion arises if applications run for a reasonable amount of time. In mobile devices such memory leaks are even more serious given the limited amount of memory available when compared to desktop computers.

Dangling references are well known to occur in centralized applications when manual memory management is used. Such errors are even more common in a classical distributed environment, i.e., in a fixed network of computers with no data replication. Thus, in a mobile environment supporting distributed applications accessing replicated objects, (correct) manual memory management is certainly harder.

In conclusion, manual memory management leads not only to applications performance

degradation and fatal errors but also to reduced programmer productivity. Thus, distributed garbage collection must be provided by mobile middleware.

Current middleware (e.g., Java and .Net) does not support distributed garbage collection. As a matter of fact, the approach taken is a simplified one, based on leases to favor liveness at the expense of safety. Objects still reachable remotely from other objects (possibly replicated) in other processes, may be discarded (reclaimed in GC terms) if they are not invoked for a certain period of time. This is clearly incorrect, as leases may expire too soon and cause dangling references; later, applications will fail when trying to access such objects. In addition, defining the leasing time is left to the application programmer, possibly leading to errors causing the violation of referential integrity of the objects graph.

Thus, when considering mobile middleware supporting data replication, the challenging requirements that mobile computing pose on distributed garbage collection (DGC), are the following: i) safety and completeness of the DGC algorithm(s) used, i.e., a real distributed GC algorithm, and not just a lease mechanism, ii) support for correct handling of replicated objects, both in local and distributed GC, withstanding data inconsistency, and iii) adaptation of local GC (LGC) algorithms to resource-constrained devices used in mobile computing. We now address the distributed garbage collection of replicated objects and files.

5.1 Distributed Garbage Collection of Replicated Objects

Several of the classical DGC algorithms, designed for function-shipping based distributed systems (i.e., with no support for replication), build upon some common elements found in algorithms such as Indirect Reference Counting (IRC) [45] or SSP Chains [46] (in particular, GC-stubs and GC-scions).

Most of these solutions [47, 48] are hybrids as 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

message	sent/received by	sent when
`NewSetStubs`	DGC/DGC	a new set of GC-stubs is available

Table 1: GC related messages.

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 (stack and static variables) 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 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.

As described in Section 3, GC-scions and GC-stubs represent incoming and out-going remote references, respectively, among objects residing in different processes, i.e., inter-process references. Thus, GC-scions and GC-stubs are created as a result of the export and import of references. A reference to an object in process P is said to be exported by P, when it is sent on a message to a process Q. A reference is said to be imported by a process Q when it is received as contents of a message delivered at Q. Note that the message just mentioned may carry one or more objects to be replicated.

Every time an object reference is exported by a process P to a process Q, the corresponding GC-scion and GC-stub must be created on P and Q, respectively. As long as an object is targeted by (at least) a GC-scion in a process, it must be preserved even when it is locally unreachable (from the local's process stack and static variables). This is due to the fact that such an object may still be invoked from other processes through an inter-process reference.

Later, due to the activity of the mutator (i.e., the application in GC terms) in the referring

event	occurs when	action taken
reference exported	replicate an object from a process	create GC-scion
reference imported	replicate an object into a process	create GC-stub
new set of GC-stubs available	local GC runs	`NewSetStubs` message sent to the processes holding the GC-scions corresponding to the previous set of GC-stubs
`NewSetStubs` message received	`NewSetStubs` message sent	compare GC-stubs set with set of GC-scions; delete GC-scions with no corresponding GC-stubs

Table 2: GC related events.

process, some remote references may disappear (and its corresponding GC-stubs) because the objects enclosing them are no longer reachable (either locally in that process, or via other remote references). Therefore, the processes holding the objects targeted by those remote references (and correspondingly, their GC-scions) are informed that the GC-stubs are no longer valid, in order to delete their counterpart GC-scions. This ensures liveness, in the sense that objects that are no longer referenced remotely, cease to be protected by the distributed GC component running in the process. Such objects are, from this moment on, at the mercy of the local GC. If they are also unreachable locally, they will be reclaimed when the next LGC occurs.

To avoid races among GC-scion creation and deletion between processes, there should not be explicit messages to delete GC-scions. In its place, there is a single kind of message that processes periodically send (the `NewSetStubs` message). The receiving processes may then detect which GC-scions have their corresponding GC-stubs no longer included in the message received, and delete them accordingly. The algorithm operation is summarized in Tables 1 and 2.

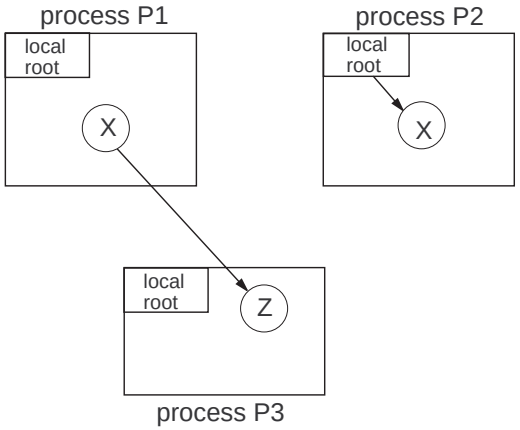

Figure 4: Safety problem of current DGC algorithms which do not handle replicated data: Z is erroneously considered unreachable.

5.1.1 DGC Correctness with Replication

In the previous section we described how DGC algorithms work in distributed systems with no replication support. However, such algorithms, designed for function-shipping based systems, are not correct when applied to replicated objects. This affects all the classical DGC algorithms that were designed for function-shipping based systems, such as Indirect Reference Counting (IRC) or SSP Chains. These algorithms are not safe in presence of replicated data, as explained next.

Consider Figure 4 in which an object X is replicated in processes P1 and P2; each replica of X is noted X_i and X_j, respectively. Now, suppose that X_i contains a reference to an object Z in another process P3, X_j points to no other object, X_i is locally unreachable and X_j is locally reachable.[2] Then, the question is: should Z be considered garbage? Classical DGC algorithms (designed for function-shipping systems) consider that Z is effectively garbage. However, this is wrong because, in a middleware object replication system, it is possible for an application in P2 to acquire a replica of X from some other process, in particular, X_i. Thus, the fact that X_i is *locally* unreachable in process P1 does not mean that X is *globally* unreachable; as a matter of fact, X_i contents can be accessed by an application in process P2 by invoking method get on

[2]Locally (un)reachability is related to (un)accessibility from the enclosing process's local root.

the corresponding proxy-in as described in Section 3.

Therefore, in a middleware object replication system, a target object Z is considered unreachable only if the union of all the replicas of the source object, X in this example, do not refer to it. This is the Union Rule introduced in Larchant [49, 50], and implemented resorting to a Union Message. This message is exchanged among processes and ensures the correct order of GC-scions creation and deletion, when objects are replicated through a number of processes. The causal delivery of such messages is needed to ensure safety of the algorithm with respect to races between the creation and deletion of GC-scions.

Larchant is the first algorithm to correctly take into account both GC-stubs, GC-scions, and the lists InPropList and OutPropList (with information about objects replicated from, and to, different processes). As described afterwards, this allows any process to update a local replica with the content of another replica from a different process (or supply its own replica to update the remote one), while one of them is still reachable in its process. This allows a mutator to access replicas (with update content resulting from local execution) that could otherwise be prematurely discarded by a non-replication-aware DGC algorithm.

5.1.2 Scalability Issues

Unfortunately, the previous solution imposes rather severe constraints on scalability. As a matter of fact, the Larchant algorithm is not scalable because it requires the underlying communication layer to support causal delivery [51].

Therefore, DGC algorithms specific to object replication systems, such as Larchant, deal safely with replication but lack scalability. To address this limitation on scalability, the requirement of causal message delivery must therefore be dropped, while still enforcing the Union Rule presented earlier. This is the approach introduced in DGC for Wide Area Replicated Memory [52].

In this algorithm, the Union Rule is preserved but it is enforced differently. This is achieved by making use of the special bits sentUmess and recUmess present in InPropList and OutPro-

message	sent/received by	sent when
`Unreachable`	LGC/DGC	object replica is reachable only from the InPropList
`Reclaim`	LGC/DGC	all object replicas are reachable only from the InPropLists

Table 3: GC messages related to replication.

event	occurs when	action taken
object replica reachable only from the InPropList	LGC runs	send `Unreachable` message to the process with the corresponding OutPropList entry; set the sentUmess bit accordingly
`Unreachable` message received	`Unreachable` message sent	set the recUmess bit accordingly; if all recUmess bits for a particular object are set, then send the corresponding `Reclaim` messages and delete the OutPropList entry
`Reclaim` message received	`Reclaim` message sent	delete corresponding InPropList entry

Table 4: GC events related to replication

pList, as shown in Table 3. Whenever a replica located in a process becomes unreachable, except from an entry in the process's InPropList, a `Unreachable` message is sent to the process holding the corresponding OutPropList entry. The sending process registers this fact by setting the sentUmess bit present in the InPropList. This prevents that this message is sent after every local garbage collection in the process. Conversely, the receiving process registers this fact by setting the recUmess bit present in the corresponding OutPropList.

Whenever an object is reachable solely from entries in the process's OutPropList, and all the corresponding `Unreachable` messages have been received, it is determined that none of the replicas is still reachable, and it is now safe to delete all of them (as a union). This is performed by lazily sending `Reclaim` messages to every process holding the replicas, followed by the deletion of the OutPropList entries. Upon reception of the `Reclaim` messages, the processes delete the corresponding InPropList entries. Finally, this enables the reclamation of all the replicas, when the next local garbage collection takes place in each of the processes.

Tables 3 and 4 present all the events related to replication concerning GC, and the cor-

responding actions taken. Thus, the last four tables combined, summarize the way GC is performed by middleware supporting object replication.

A particularly interesting case in which the DGC just presented is useful is that of web-content replication [53] with referential integrity ensured by the underlying middleware. The middleware supports HTML files (residing at web servers) that can be incrementally replicated to other computers during browsing sessions. Replicated files can be subject to editing (e.g., translation, composition with other content) and further replicated. The DGC algorithm ensures referential integrity of web content still of interest to users, i.e., files that are reachable from a root (may include bookmarks, subscription lists, etc). Therefore, the system must correctly deal with possible inconsistencies among replicas, and enforce the Union Rule already presented.

5.1.3 Other Memory Management Techniques

Mobile devices are so memory-constrained that, in some circumstances, even the memory occupied by useful reachable objects must be freed. This may occur because, at a particular instant, there are other more relevant replicas for which there is no memory available.

Freeing the memory occupied by useful objects is delicate. Given that such objects can be accessed by applications through navigation of the object graph, the middleware must still ensure the referential integrity, while freeing such memory.

This can be achieved as follows. Some object replicas (namely all of those belonging to the same class) are migrated to a nearby machine [54] named as surrogate; later, if needed, such objects are remotely invoked. To provide transparency to applications, the underlying virtual machine transforms object accesses into remote invocations. This off-loads processing demands as well as memory occupation at the extra cost of frequent remote invocations.

An alternative approach is proposed in [55, 56]. It consists in swapping-out such objects to other computers with more resources available, in particular, free memory; such replicas will be re-fetched later, if needed, and always invoked locally. In [55], a modified virtual machine manages object location and records information to assess spatial and temporal locality for

each object.

Taking into account the management of replicas described in Section 4, object clusters are natural candidates to be managed, swapped-out and swapped-back, as they have been incrementally replicated, previously, to the mobile device as a whole. This approach is adopted in [56] with small overhead, since information is kept regarding only clusters (that contain several objects) instead of each individual object. Unlike the previous approaches, it does not require the use of a specially modified virtual machine, making it rather portable.

Finally, in [57], a mechanism is proposed to perform compression of the Java virtual machine heap. To minimize application disruption, large objects (greater than 1.5 Kb) are compressed and decompressed; in addition, large array objects are broken down into smaller subobjects, each being "lazily allocated" upon its first write access.

5.2 Garbage Collection of Replicated Files

Distributed algorithms for garbage collection have also been applied in the context of replicated file systems (e.g., Ficus [20, 58], Rumor [21]). However, when compared to the case in which the middleware supports the object model, DGC is not as relevant. The reason being that, with the file model, there is no referential integrity to be maintained.

In Ficus and Rumor, a GC algorithm is used to reclaim disk space occupied by replicas of files found to have been deleted. When a process deletes a file replica, it initiates a two-phase distributed-consensus algorithm, to determine global inaccessibility of all the file replicas. Special care must be taken to resolve ambiguities between creation and deletion of different replicas of the same file (due to race conditions regarding the order of these operations).

6 Adaptability

Due to their intrinsic nature, execution environments in mobile computing suffer from great and diverse variations during applications execution. These variations can be qualitative (e.g.,

network connection or disconnection, specific devices such as printers in the device neighborhood, consistency and security constrains) or be related to quantitative aspects (e.g., amount of usable bandwidth, memory available). Applications should be able to deal with this variability; however, applications programmers should not be forced to account for every possible scenario in their coding as this is inefficient, error-prone, and limited to situations accounted for *a priori*.

Dealing with such variability can be achieved through automatic adaptation of applications. Thus, the middleware must provide flexibility for application development and runtime adaptability. Then, applications can cope with the multiple requirements and usage diversity found in mobile settings.

Adaptability is provided through the enforcement of declaratively-defined policies supported by the mobile middleware [56]. This way, policies need not be hard-coded in applications and can be deployed, enforced, and updated at any time. Mobile middleware relies strongly on the following features: i) the extensible capability to support the specification and enforcement of runtime management policies, ii) a pluggable set of basic mechanisms supporting object replication, and iii) a set of pre-defined policies to control the mechanisms previously mentioned.

The policy engine (see Figure 1) is the inference component that triggers or mediates responses to events occurred in the system; it holds a variable set of policies to be enforced in the system. The policy engine receives events generated by middleware modules and applications, evaluates policy rules and triggers events, handled by actions based on evaluation results. In particular, object replication is performed according to specified policies.

The context management module (see Figure 1) performs resource abstraction and manages properties whose values vary during execution. Resource abstraction enables the representation of physical computer resources as sets of primitive context properties. Examples include memory, connectivity, bandwidth available, etc. The actual mappings between basic/primitive resources and resource designations is performed by the context manager. Each of these re-

sources implies an architecture-dependent way of measuring. This heterogeneity is masked, to the rest of the system, by a low-level component in the context manager.

Situations such as appearing devices, discovering remote resources or application counterparts are also handled by the context manager. Detecting these situations allows the middleware and the applications to decide whether to replicate data from different sources, swap some data out, etc. In general terms, any change to the properties considered (resources, middleware state or user-defined properties) managed by the context manager can potentially trigger associated events defined by the policies loaded.

6.1 Replication Policies

A set of pre-defined policies and police-driven modules are provided to manage specific execution mechanisms. In particular, the mobile middleware must support the specification and enforcement of policies concerning the replication of objects.

Object replication is incremental and adaptive. Unless otherwise specified, it is performed transparently to applications but can also be flexibly configured. In particular, the mobile middleware must allow the specification of the following: i) the best moment to create a replica, ii) when to merge two or more replicas of the same object, iii) the amount of objects to replicate at a given time (a cluster of objects), iv) which branch of a graph should be further replicated, v) which objects should be swapped-out (i.e., dynamically replaced by a proxy-out and transfer the remaining objects to a neighboring device).

In addition, the middleware must also support the definition and enforcement of consistency related policies. Although addressed elsewhere in this book, consistency of replicas must also take into account the variability of mobile environments. This allows applications to deal, for example, with situations in which it is impossible to access the most up-to-date replica of an object but it is possible to obtain a replica slightly out-of-date that is still enough for the application to proceed. Thus, regarding object replication and consistenty, the adaptability to be supported by mobile middleware allows the specification of the following: i) alternative

sources to replicate objects from, ii) whether specific objects, clusters, or graphs, should be fetched from their authoritative home nodes, from other peers with outdated replicas, or not required at all for the application to proceed, iii) whether to cache changes made locally to the data, iv) how failures are handled, e.g., automatically weaken some of the requirements in order to be able to replicate back some of the work performed.

7 Conclusion

In this chapter we address several fundamental challenges concerning the support for replica management, memory management, and adaptability, that must be considered by mobile middleware aiming at providing seamless service access via resource replication.

Among the several programming paradigms available, we focused on two: object model and file model. The first is widely used and is highly flexible; the second is well known both by users and applications programmers, and is supported by most operating systems. While the object model allows applications to navigate on a graph of objects, the file model does not support such concept. As explained in this chapter, this difference has important consequences on replica management and memory management.

Regarding mobile middleware supporting the object model, we present an archetypal architecture and describe how object replication can be achieved taking into account the limited amount of memory available in mobile devices (when compared to desktop computers). The mechanism of incremental replication provides the needed flexibility for applications to deal with the variability of network availability/bandwidth and amount of free memory.

Concerning the file model, we address how files can be replicated while minimizing the space occupied taking advantage of replicas' contents similarities. This mechanism also contributes to reduce the amount of network communication required for replica creation and updating.

Deciding which and when files must be replicated is a hard problem that still raises interest-

ing research issues. We mention some relevant work in the area of file hoarding but there is still too much intervention needed from the user. Ideally, the mobile middleware should discover automatically which files should be replicated and when, so that users are never prevented from doing their work.

Concerning distributed garbage collection, this chapter presents the most relevant algorithms that are able to deal correctly with replicated objects. However, they all lack the capability to reclaim distributed cycles of garbage replicas, i.e., the algorithms are not complete. This is a hard problem on which more research is needed. Lessons may be learned from recent complete DGC algorithms [59, 60, 61, 62] developed for function-shipping systems, i.e., without replication support. The first of such approaches, able to reclaim cycles of replicated garbage objects, is presented in [63].

The garbage collection in middleware supporting the file model aims at reclaiming disk space occupied by deleted files. Thus, the problem being tackled is that of disk space management; there is no referential integrity to be ensured as it happens with the object model.

Finally, we address the issue of adaptability, i.e., how the mobile middleware allows applications to control and adapt to the resources they use. In particular, we focus on replication policies whose relevance results from the high variability of mobile environments in terms of network quality and memory available. A basic aspect is the need to clearly separate policies from mechanisms and allowing such policies to be dynamically instantiated. Although some solutions have been proposed by several projects, there is still much research to be done so that applications programmers may focus on the application logic without having to bother with system level issues.

A successful mobile middleware platform has to deal with many other aspects besides those addressed in this chapter. However, we believe that replica management, memory management and adaptability are among those that are crucial. This chapter provides solutions to such issues and highlights some research topics deserving future attention.

Acknowledegments We thank the institutions that have supported the authors: FCT (Fundação para a Ciência e a Tecnologia, Portugal) and Microsoft Research.

References

[1] M. Satyanarayanan, "Fundamental challenges in mobile computing," in *Symposium on Principles of Distributed Computing*, pp. 1–7, 1996.

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

[3] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira, "Disconnected operation in thor object-oriented database system," in *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, (Santa Cruz, CA), December 1994.

[4] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," in *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pp. 190–201, ACM Press, 2000.

[5] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," *SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 190–201, 2000.

[6] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 5, pp. 190–201, 2000.

[7] U. Çetintemel, P. J. Keleher, B. Bhattacharjee, and M. J. Franklin, "Deno: A decentralized, peer-to-peer object-replication system for weakly connected environments," *IEEE Trans. Comput.*, vol. 52, no. 7, pp. 943–959, 2003.

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

[9] P. Ferreira, L. Veiga, and C. Ribeiro, "Obiwan - design and implementation of a middleware platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14(11), pp. 1086–1099, November 2003.

[10] L. Veiga, N. Santos, R. Lebre, and P. Ferreira, "Loosely-coupled, mobile replication of objects with transactions," in *Workshop on Qos and Dynamic Systems. 10th IEEE International Conference On Parallel and Distributed Systems(ICPADS 2004)*, 2004.

[11] N. Santos, L. Veiga, and P. Ferreira, "Transaction policies for mobile networks," in *5th IEEE International Workshop on Policies for Dist. Systems and Networks(Policy 2004)*, 2004.

[12] C. Pairot, P. García, and A. F. G. Skarmeta, "Dermi: A decentralized peer-to-peer event-based object middleware," in *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 236–243, IEEE Computer Society, 2004.

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

[14] D. Hagimont and F. Boyer, "A configurable RMI mechanism for sharing distributed Java objects," *IEEE Internet Computing*, vol. 5, Jan. 2001.

[15] M. Butrico, H. Chang, A. Cocchi, N. Cohen, D. Shea, and S. Smith, "Gold rush: Mobile transaction middleware with java-object replication," in *3rd Usenix Conference on Object-Oriented, Technologies*, Usenix, 1997.

[16] M. Haahr, R. Cunningham, and V. Cahill, "Towards a generic architecture for mobile object-oriented applications," Dec. 2000.

[17] J. Siegel, *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc, 1996.

[18] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 3–25, 1992.

[19] M. Satyanarayanan, "The evolution of coda," *ACM Trans. Comput. Syst.*, vol. 20, no. 2, pp. 85–124, 2002.

[20] G. J. Popek, R. G. Guy, T. W. Page, Jr., and J. S. Heidemann, "Replication in Ficus distributed file systems," in *Proceedings of the Workshop on Management of Replicated Data*, pp. 20–25, University of California, Los Angeles, IEEE, November 1990.

[21] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek, "Rumor: Mobile data access through optimistic peer-to-peer replication," in *ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining*, pp. 254–265, Springer-Verlag, 1999.

[22] D. Ratner, P. Reiher, G. J. Popek, and G. H. Kuenning, "Replication requirements in mobile environments," *Mob. Netw. Appl.*, vol. 6, no. 6, pp. 525–533, 2001.

[23] D. Ratner, P. Reiher, and G. J. Popek, "Roam: a scalable replication system for mobility," *Mob. Netw. Appl.*, vol. 9, no. 5, pp. 537–544, 2004.

[24] J. N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[25] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch, "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, 8-9 1994.

[26] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 172–182, ACM Press, 1995.

[27] D. B. Terry, K. Petersen, M. Spreitzer, and M. Theimer, "The case for non-transparent replication: Examples from bayou," *IEEE Data Eng. Bull.*, vol. 21, no. 4, pp. 12–20, 1998.

[28] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos, "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)*, 2003.

[29] A. Sinha, "Client-server computing," *Commun. ACM*, vol. 35, no. 7, pp. 77–98, 1992.

[30] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 335–371, 2004.

[31] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, 1997.

[32] M. Shapiro, "Structure and encapsulation in distributed systems: the proxy principle," in *Proc. of the 6th International Conference on Distributed Systems*, (Boston), pp. 198–204, May 1986.

[33] K. W. Mary Campione, *The Java Tutorial, Second Edition: Object Oriented Programming for the Internet*. Sun Java Series, Addison Wesley Professional. ISBN 0-201-31007-4.

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

[35] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite network file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 134–154, 1988.

[36] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Symposium on Operating Systems Principles*, pp. 174–187, 2001.

[37] J. Barreto and P. Ferreira, "A replicated file system for resource constrained mobile devices," in *International Conference on Applied Computing (IADIS)*, 2004.

[38] J. Barreto and P. Ferreira, "A highly available replicated file system for resource-constrained windows ce .net devices," in *3rd International Conference on .NET Technologies*, 2005.

[39] N. I. of Standards and Technology, "Fips pub 180-1: Secure hash standard," tech. rep., Gaithersburg, 1995.

[40] M. Rabin, "Technical report tr-15-81: Fingerprinting by random polynomials," tech. rep., Center for Research in Computing Technology, Harvard University, 1981.

[41] G. Chen and D. Kotz, "A survey of context-aware mobile computing research," Tech. Rep. TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[42] C. Tait, H. Lei, S. Acharya, and H. Chang, "Intelligent file hoarding for mobile computers," in *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, (New York, NY, USA), pp. 119–125, ACM Press, 1995.

[43] G. H. Kuenning and G. J. Popek, "Automated hoarding for mobile computers," in *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 264–275, ACM Press, 1997.

[44] G. H. Kuenning, W. Ma, P. Reiher, and G. J. Popek, "Simplifying automated hoarding methods," in *MSWiM '02: Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*, (New York, NY, USA), pp. 15–21, ACM Press, 2002.

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

[46] M. Shapiro, P. Dickman, and D. Plainfossé, "SSP chains: Robust, dist. references supporting acyclic garbage collection," Rapport de Recherche 1799, Institut National de Recherche en Informatique et Automatique, Rocquencourt (France), Nov. 1992. http://www-sor.inria.fr/SOR/docs/SSPC_rr1799.html.

[47] S. E. Abdullahi and G. A. Ringwood, "Garbage collecting the internet: a survey of distributed garbage collection," *ACM Comput. Surv.*, vol. 30, no. 3, pp. 330–373, 1998.

[48] D. Plainfossé and M. Shapiro, "A survey of distributed garbage collection techniques," in *Proc. Int. Workshop on Memory Management*, (Kinross Scotland (UK)), Sept. 1995.

[49] P. Ferreira and M. Shapiro, "Larchant: Persistence by reachability in distributed shared memory through garbage collection," in *Sixteenth International Conference on Distributed Computer Systems*, (Hong Kong), 1996.

[50] P. Ferreira and M. 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.

[51] D. R. Cheriton and D. Skeen, "Understanding the limitations of causally and totally ordered communication," in *Proc. of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 44–57, 1993.

[52] A. Sanchez, L. Veiga, and P. Ferreira, "Distributed garbage collection for wide area replicated memory," in *Proc. of the Sixth USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'01)*, (San Antonio (USA)), Jan. 2001.

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

[54] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, p. 43, IEEE Computer Society, 2002.

[55] D. Chen, A. Messer, D. Milojicic, and S. Dwarkadas, "Garbage collector assisted memory offloading for memory-constrained devices," in *Fifth IEEE Workshop on Mobile Computing Systems & Applications*.

[56] L. Veiga and P. Ferreira, "Poliper : Policies for mobile and pervasive environments," in *3rd Workshop on Reflective and Adaptive Middleware. In 6th ACM International Middleware Conference*, (Toronto, Canada), October 2004.

[57] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko, "Heap compression for memory-constrained java environments," in *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 282–301, ACM Press, 2003.

[58] D. Ratner, P. L. Reiher, G. J. Popek, and R. G. Guy, "Peer replication with selective control," in *MDA '99: Proceedings of the First International Conference on Mobile Data Access*, pp. 169–181, Springer-Verlag, 1999.

[59] H. C. C. D. Rodrigues and R. E. Jones, "Cyclic dist. garbage collection with group merger," in *Proc. of 12th European Conf. on Object-Oriented Programming, ECOOP98* (E. Jul, ed.), Lecture Notes in Computer Science, (Brussels), pp. 249–273, Springer-Verlag, July 1998. Also UKC Technical report 17–97, December 1997.

[60] F. L. Fessant, "Detecting distributed cycles of garbage in large-scale systems," in *Conference on Principles of Distributed Computing(PODC)*, 2001.

[61] L. Veiga and P. Ferreira, "Complete distributed garbage collection, an experience with rotor," *IEE Research Journals - Software*, vol. 150(5), oct 2003.

[62] L. Veiga and P. Ferreira, "Asynchronous complete distributed garbage collection," in *19th IEEE International Parallel and Distributed Processing Symposium*, (Denver, CO, USA), april 2005.

[63] L. Veiga and P. Ferreira, "A comprehensive approach for memory management of replicated objects," technical report rt/07/2005, INESC-ID Lisboa, april 2005.

Paulo Ferreira is Associate Professor at the Computer and Information Systems Department at the Technical University of Lisbon, Portugal. In 1996, he received his PhD degree in Computer Science from Université Pierre et Marie Curie. He is a researcher at INESC-ID since 1986 where he leads the Distributed Systems Group. His research interests include operating system and middleware support for large-scale and mobile distributed data sharing. He is author or co-author of more than 50 peer-reviewed scientific communications and he has served on the program committees of several international journals, conferences and workshops in the area of distributed systems.

Luís Veiga received his BsCE (1998) and MSc degrees (2001) in Computer Engineering, from the Technical University of Lisbon (Instituto Superior Técnico), Portugal. He is a Lecturer and PhD candidate in the Computer and Information Systems Department. He is a Researcher at INESC-ID (Distributed Systems Group) since 1999 having participated in projects such as Mnemosyne, MobileTrans, OBIWAN, DGC-Rotor, UbiRep. His research interests include distributed systems, memory management for distributed and mobile computing, replication, distributed garbage collection, mobile middleware. He has authored or co-authored 15 peer-reviewed scientific communications in workshops, conferences, journals; and served as reviewer in two international conferences.