Limited Copies and Leased References for Distributed Persistent Objects

Susan Spence

Submitted for the Degree of Doctor of Philosophy
Department of Computing Science
University of Glasgow
March 2000

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Contents

List of Figures

Abstract

As businesses become global organisations and as e-commerce opens up markets to customers across the Internet, demand grows for increasingly ambitious distributed software applications and platforms. Where these applications run over potentially huge collections of data, sophisticated management of data storage and communication is required. There is a need for well-integrated persistence and distribution support that considers the implications for long-term maintenance of valuable persistent data.

Orthogonal persistence is intended to ease the programmer's job by providing support for data management that is integrated with a programming language. The simplicity of the orthogonal persistence model argues for its use in distributed systems, in order to make life simpler for the application programmer. PJRMI is an implementation of Java RMI for the orthogonally-persistent PJama platform. This dissertation addresses two problem areas raised by combining orthogonal persistence with support for distributed applications. These problem areas are illustrated by PJRMI.

The first problem is raised as a consequence of attempting to provide the illusion of a persistent connection between stores. Distribution-related errors easily break this illusion. In an open system, it can be difficult to determine when an object should become persistent by remote reachability. In the long term, persistent references to remote objects threaten the maintainability of the persistent stores involved.

A solution has been implemented to address the problems raised by maintaining persistent references between distributed stores. Greater autonomy of individual stores is achieved by limiting remote access to objects to a duration of time associated with a specific distributed application's lifetime. Within the application's lifetime, the benefits are retained of persistence of inter-store references for resilience.

The second problem is encountered when copying object graphs between stores. Large object graphs tend to build up in persistent stores over time. Copying such large object graphs can be prohibitively expensive in terms of resources and performance. A programmer may assume that the size of graph they are copying is acceptable, based on their knowledge of

a system in its infancy. However, the problem is that, in a long-lived system, their assumptions may be challenged, since the size of an object graph and the context in which it is used are more likely to change during a persistent object graph's lifetime. The combination of a typically statically-defined policy for passing objects to remote sites and programmer assumptions that fail to take into account the lifetime of an object can also result in other problems. These problems include failure to support different requirements on remote use of the same object graph by different applications during that object graph's lifetime.

A solution has been implemented to address the problems raised by remote copying of large object graphs. Flexibility of control over such copying is achieved. Separation of policy from object definition ensures flexibility. Choice of object-copying policy for a specific distributed application's lifetime provides control, while ensuring it is adaptable to changes in size of persistent object graphs over their lifetime and to changes in the context in which these graphs are used.

Acknowledgements

This dissertation has been accomplished with much support from my supervisor, partner, family, friends and colleagues. I am very grateful to them all. Special thanks are due to those named below.

Malcolm Atkinson: for his energy, enthusiasm, encouragement as my supervisor and for his ability to keep his researchers in contracts and interesting, challenging jobs.

Satnam Singh: for waiting so patiently on another continent for me to finish this PhD and for agreeing to be my husband.

My parents Graham and Irene Spence: for their love and support; they are always there for me.

My sister Alison Campbell: for her love and encouragement and especially for the PhD monster decoy. It worked - I've escaped!

Carol Emslie: for our enduring friendship, particularly through the university years of relaxed flat-sharing in Dowanhill Street.

Mick Jordan: for supporting my internships at Sun Microsystems Laboratories.

Quintin Cutts: for his role as second supervisor. I enjoyed comparing notes on time management.

Peter Dickman: for taking on the unofficial role of third supervisor.

All my friends and colleagues in the Department of Computing Science at the University of Glasgow: through my four years as an undergraduate and seven and a half years as a research assistant. It's the people of this excellant department that have made me stay around as long as I have.

Huw Evans: for his friendship, thorough proof-reading, leading me astray with whisky and good memories of trips, especially to Paris and Pisa, to appreciate their fine food and wine, purely in the line of duty. I appreciated even his most in-seine jokes.

Tony Printezis: I have greatly enjoyed the fruits of his enthusiasm for cooking; even though it has never quite matched his enthusiasm for C hacking and garbage collection.

Craig Hamilton: who always has an entertainingly sharp word to say on some of our favourite PJama project conversational subjects.

Others I have worked with on the PJama project, including Laurent Daynès.

Users of PJRMI: who kindly provided me with feedback on using PJRMI in their work and cooperated, sometimes very patiently over weeks at a time, in helping me to fix bugs with PJRMI and PJama in general.

This work has been supported by the PJama project, funded by Sun Microsystems Laboratories and the EPSRC.

Apologies to those readers who, like Huw, do not always appreciate the verbose nature of my writing; it could admittedly be considered, as Stephen Fry would say, rather pleonastic or sesquipedalian or ...

Chapter 1

Introduction

According to a recent article in the Financial Times [FT98], "the Internet will inevitably become the dominant medium for the global economy". This is backed up by USA Today, which reports that "The Internet economy generated $301 billion in revenue last year" and that "The Internet economy is doubling every nine months" [Bel99]. A quarterly report on Internet Economy Indicators, by the University of Texas Center for Research in Electronic Commerce [II00], provides many more fascinating statistics on this subject.

As more businesses become global organisations and as e-commerce opens up markets to customers across the Internet, demand grows for increasingly ambitious distributed software applications and platforms. Where these applications run over potentially huge collections of data, sophisticated management of its storage and communication is needed, to handle data access across wide area networks between, for example, the departments of an organisation around the world, as well as across local area networks within one site of an organisation. Sun Microsystems, with offices in 150 countries, is a good example of a global business that increasingly runs product and employee information and administration systems over wide area networks [Sun99].

Consequently, programmers need flexible, reliable platforms that will ease both development and long-term maintenance of these distributed applications and their associated data management.

Orthogonal persistence is intended to ease the programmer's job by providing support for data management that is integrated with a programming language. By automating the storage of data and propagation of its updates to disk, the application programmer's job is simplified, leaving them to focus on the coding of the application itself. The PJama project has designed and implemented orthogonal persistence for the object-oriented programming language Java [ADJ+96, JA98]. The type-safety of Java makes it an appropriate language

for integration with orthogonal persistence. Strong typing is crucial for maintaining consistent graphs of objects in stable storage. The commercial viability of Java, its purported platform neutrality and the current popularity of object-oriented programming enables the PJama project to make its research available and attractive to a wide audience.

Providing orthogonal persistence of objects within a single address space is well-understood. The challenge, partly addressed by the work in this dissertation, is how to address the issues raised by combining orthogonal persistence with support for distributed applications.

1.1 Overview of Problems Combining Persistence and Distribution

The use of orthogonal persistence in a distributed system has a number of implications. This dissertation focusses on dealing with these implications in two subject areas.

1.1.1 Implications of Dependencies Between Stores

Orthogonal persistence, as implemented for the PJama platform and summarised in chapter 2, maintains a consistent and stable state of the objects that become reachable, via references, from objects identified as roots of persistence. Within one process running over a persistent store, it is possible to guarantee the consistent, stable state of persistent objects.

The simplicity of the orthogonal persistence model argues for its use in distributed systems, in order to make life simpler for the application programmer. Despite the inherently transient nature of connections between distributed objects, the illusion of a persistent connection can be provided, as demonstrated by the support for persistent remote method invocation for Java (PJRMI) described in chapter 3.

However, such attempts to extend orthogonal persistence, from a single process to the less reliable and less controllable world of a distributed system, sacrifice the guarantees on consistency (and the integrity of object references, in particular) in the persistent stores involved. It is unrealistic to assume that, just because a reference to a remote object has been made persistent, it will always be possible to access the remote object successfully. Distribution-related errors caused by process crashes and network delays or failures easily break the illusion of a persistent connection.

Another challenge, for support of persistent references to remote objects, is that it can also be difficult to ensure that the remotely-referenced object exists as long as it is required. Extending persistence by reachability across a distributed system implies that if an object becomes persistent and it holds a reference to a remote object then the remote object must

become persistent too. It can be difficult to determine when and how an object should become persistent by remote reachability though, as described in chapter 6.

There is also a long-term problem with persistent connections between distributed objects: they threaten the maintainability of the persistent stores involved, by decreasing autonomy of an individual store's data management. A store does not have the control to maintain a consistent state over its objects and to garbage-collect those that it no longer wishes to contain if it is obliged to provide remote access to objects for as long as references are held to them from other stores. By the same token, a store does not have control over the integrity of its references when it holds a reference to an object in a remote store, making it dependent on the remote store for its own referential integrity.

1.1.2 Problems with Copying Object Graphs Between Stores

The trend for remote object access in distributed programming is currently moving away from the model of passing objects solely by reference (as espoused by DCOM and, until recently, CORBA) to one where objects can also be copied between processes. Thus, having considered some of the implications of managing references between persistent, distributed objects, focus is now placed on how to manage the copying of persistent object graphs across a distributed system, when such object-copying is required by an application. (For clarification: the issue of object migration is not one of the topics of this dissertation.)

The introduction of persistence into a distributed application changes assumptions about how objects are used in a distributed system. For a distributed application with no persistence support, the programmer is likely to make the assumptions that the object graphs passed by copy between processes will be small and always used in the same way, in the same context.

However, like traditional databases, a persistent object store is often populated incrementally, with the intention of maintaining it over months or years. Large object graphs can build up in persistent stores over time. Thus, for example, an application that remotely accesses a persistent object graph by making a deep copy of it may be able to do so efficiently during executions early in the lifetime of the store, but it may find that such copying has prohibitive costs or that it even becomes error-prone, as the object graph grows. The long lifetime of the store increases the likelihood that the same persistent object graphs may be used by different applications. It also increases the likelihood that the same persistent object graphs may be used in different distributed environments.

Given that current practice is for the policies for passing objects between processes to be defined statically, tied to the object's class definition, there is a lack of flexibility for adapting the copying of persistent object graphs between processes to cope with their size and the

context in which they are used, when in fact both may change during the lifetime of a store.

1.2 Realistic Solutions for Persistence and Distribution

The emphasis on the solutions proposed in this dissertation, for dealing with the problems above, is that they be realistic, rather than idealistic. Having examined the approaches of related work to these problems, presented in chapter 4, and found them wanting, the author's solutions address the two problem areas as summarised below.

1.2.1 Limiting Dependencies Between Stores

The application programmer must choose which of two issues is most important for their persistent, distributed application: a simple model of programming with automated storage of objects, even when those objects represent objects in a remote store, or a reliable, consistent, local persistent store. Realistically, because of the intrinsic lack of reliability in a distributed system, they cannot rely on having both.

To run a distributed application with reliable, consistent persistent stores, it is necessary to ensure that no references to remote objects ever become reachable from a persistent object and to ensure that no process that uses an object remotely is long-running, in order to limit the obligation of the store providing remote access to the object.

On the other hand, to take advantage of the orthogonal persistence model for applications running over distributed persistent stores, the application programmer must make a trade-off between the simplicity of using distributed objects that can become persistent and the consequent lack of reliability and consistency in their persistent stores.

Support has been developed for a compromise, described in chapter 6, that provides the benefit of persistent, distributed objects within the lifetime of a distributed application. (The lifetime of a distributed application is the time for which a group of distributed application programs run until the application is completed; this run may span multiple process executions, across store shutdowns and restarts.) A conservative position is taken on the persistence of remotely-accessible objects for the duration of an application's lifetime. The compromise involves introducing time limits, appropriate to the duration of a given application's lifetime, on the remote accessibility of objects and on the usability of references to remote objects. The long-term usability of references to remote objects is traded off against the increased autonomy of persistent stores, with the intention of increasing the stores' long-term maintainability.

1.2.2 Policies for Flexible Object Graph Copying between Stores

In order to avoid making fixed assumptions about the copying of object graphs between distributed processes, it is necessary to avoid statically defining the copying policy within the class of an application object. Chapter 8 describes how a separation of architectural issues is achieved by instead specifying an object-copying policy in its own class, separately from the classes of a particular application and those of the objects it uses. A wrapper class is then used to apply a particular object-copying policy for the lifetime of an application. The details of the implementation can be found in chapter 9.

For evaluation, a number of object-copying policies have been developed and tested with applications, as described in chapter 10. Policies for limiting the copying of large object graphs between processes are demonstrated; different policies are successfully applied to the same persistent object graphs used by different applications; and different policies show adaptability to the changing scale of network for different executions of the same application.

1.3 Thesis Statement

Existing platform support for orthogonal persistence of objects and distribution of those objects over wide area networks is not sufficiently integrated or flexible. This dissertation addresses two important issues raised by providing such integrated support in an open, persistent system.

Supporting referential integrity for the lifetime of persistent references to remote objects places unrealistic obligations on the stores containing the referenced objects. A tradeoff is made between resilience of inter-store references and maintainability through autonomy of individual persistent stores. This is done by combining support for persistent references to remote objects in the short-term, with appropriately-set timeouts on access to the remotely-referenced objects in the long-term.

Where the passing of objects by copy between persistent stores is required, support is needed to avoid unnecessary or prohibitively-large serialisations of persistent object graphs. A number of object-copying policies have been developed. For evaluation, and to illustrate how the separation of class definition from object-copying policy can be achieved, experiments have been performed with a variety of applications. These applications can use the same object graphs in different ways and in diverse distributed environments, given an appropriate object-copying policy.

1.4 The Guided Tour

Chapter 2: Orthogonal Persistence
Defines orthogonal persistence and introduces the PJama project's implementation of it for the object-oriented programming language Java.

Chapter 3: Persistent Remote Method Invocation (PJRMI)
Describes support for maintaining the illusion of persistent connections between distributed objects; developed for the PJama platform by the author. PJRMI forms the basis for exploration of the problems raised in the author's thesis and experimentation with the proposed solutions.

Chapter 4: Approaches of Related Work
Examines the approaches of related work to the specified problems raised by combining persistence and distribution support.

Chapter 5: Research Issues to be Addressed
Summarises the problems that have been raised and existing approaches taken to deal with them. The scene is set for addressing each of the two problems. The rest of the dissertation is presented in two parts: the first part, in chapter 6, presents the author's solution to the problem raised in section 1.1.1; the second part then presents the author's solution to the problem raised in section 1.1.2.

Chapter 6: Persistence by Reachability across a Distributed System
Explores the issues associated with extending persistence by reachability across a distributed system. Presents leases, set on remote use of persistent objects for the duration of a distributed application's lifetime, as a solution that compromises on reliability in favour of greater store autonomy.

Chapter 7, 8, 9, 10: Object Copying Policies:
Introduction, Design, Implementation and Evaluation
States the motivations and assumptions behind the use of object-copying policies for persistent applications. Presents the design and implementation of these policies. The policies ensure adaptability, over time, for the copying of objects between persistent stores to deal, in particular, with the problem of how to handle large graphs of persistent objects in a distributed system. The policy support is shown to be adaptable in use with several applications.

Chapter 11: Future Work
Describes challenges for future work in the area of persistence and distribution.

Chapter 12: Conclusion
Summarises achievements of the author's work and presents the conclusions.

Chapter 2

Orthogonal Persistence

Orthogonal persistence [AM95] integrates data management into the support for a programming language, so that it no longer pervades application code. In traditional database applications, data management commands, in SQL for example, are embedded throughout the application code, explicitly managing the movement of data between memory and the database on disk. In comparison, applications using orthogonal persistence usually need only a few lines at the beginning of an application to indicate which objects will persist. Thereafter, the application programmer can focus solely on the application task, while the persistent system automatically manages application data storage and updates transparently.

Support for orthogonal persistence in an object-oriented language, is required, as described in [AM95], to meet the following criteria:

- *Persistence is orthogonal to type*: The lifetime of an object does not depend on its type. Thus, there is no restriction on which types can be made persistent.
- *Persistence independence*: The application code for creating and using objects is always the same; i.e. it's independent of the lifetime of the objects themselves. The point here is that there is no specialised code for creating persistent objects, that is different from that for creating objects that will not persist beyond the current program execution.
- *Simple persistence identification*: A simple mechanism is used to identify those objects which are to persist beyond the program execution in which they are created. Conforming to the criteria above, this mechanism must be independent of the type system.

2.1 Orthogonal Persistence for Java

The PJama project has produced a specification for Orthogonal Persistence for Java (OPJ) [JA99] and a number of releases of the PJama implementation of orthogonal persistence for Java [ADJ+96, JA98] have been made for research and evaluation purposes. A PJama release includes a Java Virtual Machine modified for support of persistence and the Java classes that provide the PJama API.

Applying the orthogonal persistence criteria, in OPJ an object of any Java class may persist; including the `Class` objects themselves, threads, windows, etc. Persistence by reachability is used to identify persistent objects. An object registered by name using the PJama API is treated as a root of persistence; there are usually only a small number of these root objects per store – typically one per application. Other objects that become reachable, directly or indirectly, from a persistent root will themselves become persistent. These are referred to as "persistence reachable" objects. Ensuring that all objects that are persistence reachable do become persistent guarantees referential integrity: a persistent object should never be left holding a dangling reference.

The type-safety of Java makes it an appropriate language for integration with orthogonal persistence. Strong typing helps to ensure the referential integrity of object graphs within a persistent store, which is crucial for maintaining persistence by reachability reliably. As long as an object is reachable from a persistent root, PJama automatically maintains both its data and code on stable storage. The commercial viability of Java also enables the PJama project to make its research available and attractive to a wide audience.

The work for this dissertation has been done with successive releases of PJama integrated with 1.1.x and 1.2.x Classic versions of the Java Development Kit (JDK), the latest of which is PJama version 0.5.7.13 [PJR98]. A second generation implementation of OPJ has subsequently been released with a simpler API and more scalable store implementation [PAD+98b, PAD98a]. Integrated with JDK 1.2 for Solaris production release[1], it is available from Sun Microsystems Laboratories as PJama version 1.5.1 and upwards [For00].

2.2 PJama: an Open Persistent System

If PJama was a closed, persistent system, where everything in a program was under the control of the persistent system, as in Napier88 [MCC+99], the state of all supported data types would be known and could be made persistent but no other state external to the system

[1]Renamed "SunLabs Virtual Machine for Research (ResearchVM)" when re-targeted to purely research purposes in autumn 1999.

would be handled. Napier88 supports interaction with some system-level entities: files, windows and sockets; but this support is built into Napier's implementation. Napier88 cannot interact with any technology that is not specifically managed in its implementation.

Being an open, persistent system enables PJama programs to make use of many other technologies, rather than making it necessary to implement these technologies entirely in the PJama platform. To enable such openness, PJama provides a way for programmers to specify the extra, specialised support for dealing with external technologies.

This is where the effects of running in an open, persistent system are felt. Java classes can use facilities such as windowing toolkits and socket connections, which are inherently transient and outside the control of an open persistent system. Since referential integrity cannot be maintained between persistent objects and the external resources that they reference, PJama's hooks for specifying extra, specialised support must be exploited to deal with them.

One issue, of relevance to distribution support for PJama, which currently challenges PJama's claim to orthogonal persistence, is the handling of threads. The aim is that support will be provided for persistent threads in the future, but technical difficulties currently prevent its implementation. Thus, PJama's hooks for managing external technologies must currently also be used to deal with threads.

2.3 Managing Externalities

Java objects can be created which represent entities that are intrinsically transient in, or external to, the PJama platform. Such objects may represent, for example, sockets, files, windows or threads. Although the objects may become persistent by reachability, the things that they represent will not actually be usable across multiple program invocations, because they are not under the control of PJama. Thus, extra support is needed for PJama to try to re-establish the state of these objects as required, after they have become reachable from persistent roots.

Two mechanisms are used by PJama to manage, at key points in the execution of a persistent application, the state of these objects, which may be viewed as persistent by a persistent application, but which are actually objects external to the persistent system.

Firstly, fields of a class can be marked as being transient[2] using a static method of the PJama

[2]See [PAJ99] on the differing interpretations of the definition and handling of `transient` fields in Java and PJama.

API class `org.opj.utilities.PJSystem`.

```
public final static void markTransient(Class clazz, String fieldName)
```

PJama interprets any field marked transient in this way as a field which should not persist, even when the object which contains it is made persistent. Instead, this field is set to a default value (null or zero) on store restart.

Secondly, an instance of the PJama API class `org.opj.store.PJActionHandler` can be used to, for example, open and close sockets and files, open and close GUI windows and start and stop threads associated with objects in the store. `PJActionHandlers` are registered with a `org.opj.store.PJActionManager`, which ensures that they are executed at significant points in a persistent program's execution: on *startup*: just before program execution begins when re-opening a persistent store; on *stabilisation*: just before a user-initiated stabilisation (checkpoint) of reachable object state to persistent storage during program execution or on *shutdown*: just before the implicit stabilisation at the end of a program's successful execution.

The use of these mechanisms, for handling socket connections and threads associated with remote method invocations, is described in detail in chapter 3. For more on the usage of `PJActionHandlers` in general, see [JA99]. Documentation on the use of `PJActionHandlers`, with examples, can be found on the `javadoc`-generated HTML page for the `PJActionManager` interface. The documentation for the `PJActionManager` and its associated classes is part of the PJama API documentation distributed with the PJama software releases, up to and including PJama version 0.5.20.2. The support for `PJActionHandlers` has been redesigned and reimplemented for the second generation of PJama.

Chapter 3

Persistent Remote Method Invocation (PJRMI)

Remote method invocation (RMI) is the object-oriented equivalent of RPC, the well-known procedural model of inter-process communication [BN84]. Java RMI is an example of an RMI implementation [RMI98]. It supports the calling of a method of an object instantiated in one Java Virtual Machine (JVM), from the code of another object, instantiated in a different JVM. The two JVMs involved in the call may be on the same or on different host machines.

The use of standard Java RMI in the context of PJama becomes problematic when remotely-invokable objects and objects holding references to them from other VMs become persistent by reachability. This is because, without additional support, they will be unusable after store restart.

As the context in which to investigate distribution issues for a persistent system, an implementation of RMI enhanced for PJama (PJRMI) has been developed [1]; providing additional support to ensure a working and understandable usage of persistent RMI objects. It addresses the need for maintenance of the same object identity for a persistent RMI object across multiple program executions and handles externalities, such as socket connections in the persistent context.

PJRMI is described in detail in this chapter, since it forms the base for the research presented in the rest of this dissertation. Relevant details of Java RMI are introduced in section 3.1. The problems of using Java RMI in the context of an orthogonally-persistent system are described in section 3.2. The solutions supported by PJRMI are presented in section 3.3. This is followed by the details of the PJRMI implementation in section 3.4. The chapter is

[1] PJRMI was developed with versions of PJama using JDK1.1.x; then ported later to PJama using JDK1.2.x.

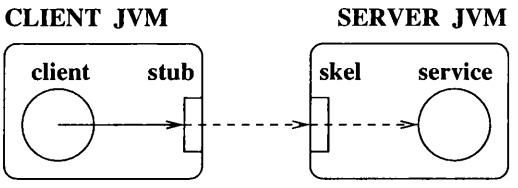

Figure 3.1: Objects in an RMI call

concluded with section 3.5 on how users have employed PJRMI for their applications and feedback from their experiences. The tutorial used to introduce users to PJRMI is included as appendix A.

3.1 Java RMI

The basics of Java RMI are described in this section, including details of the implementation which are relevant to discussions later in this chapter.

A number of objects are usually involved in an RMI call, as illustrated in figure 3.1.

- A remotely-invokable object provides a service: it implements a specified interface to those of its methods that can be called remotely. RMI mechanisms are used to "export" it in order to make it available for remote use.
- A client object obtains a reference to the remotely-invokable object.
- A stub (proxy) object represents the remotely-invokable object in the client's JVM. The stub holds information on the location and identity of the object it represents.
- A skeleton object forwards calls, received at the server-side from the stub, to the remotely-invokable object, and returns the results of these calls back through the stub to the caller.

In a standard RMI program, a remotely-invokable object is created and usually made available until the program is terminated. From the point at which it is exported for remote use, a thread listens for incoming connections on its behalf; this daemon thread runs indefinitely, or at least until its host JVM is shut down.

Objects in another JVM wishing to use the remotely-invokable object can obtain references to it in one of two ways. Usually, clients obtain the references from other application objects. For bootstrapping purposes, Java RMI also includes support for a name service called

the RMI Registry. This is run on the same host, usually as a separate process. A remotely-invokable object can be registered by name with the Registry. Subsequently, clients anywhere on the network can look up the remotely-invokable object by name to obtain a reference to it.

Client objects treat the obtained reference as a direct reference to the remotely-invokable object. However, they are actually given a reference to a local stub for that object, created automatically in the client's JVM. Calls made by the client to the remote object's interface are actually invoked on its local stub. The RMI implementation then uses a socket connection to send these calls to the JVM hosting the remotely-invokable object, where dispatcher code in the corresponding skeleton object invokes the appropriate method and returns the result, again via the stub, to the client. The client thread making the RMI call is blocked until the method has been invoked remotely and the call returns.

The object to be invoked from a client is identified in the stub object by: the host and port number where a thread is listening for incoming connections on behalf of the remotely-invokable object, plus an object identity composed from the identity of the JVM and a count incremented for each object identity generated in that JVM. Thus, the identity in the stub identifies an object in a specific JVM on a specific host.

Java RMI also includes support for Distributed Garbage Collection (DGC). This is based on DGC for Network Objects [BEN+93]. This DGC system uses reference listing: each JVM supporting remotely-invokable objects maintains a list of client JVMs that hold references to them. For each client JVM in that list, another list is kept of the specific remotely-invokable objects which are referenced by that client. As far as DGC is concerned, a reference from a client JVM is only valid until: it is no longer reachable and is garbage-collected; the client has failed to contact the server within a server-specified lease period of time; or the client has terminated.

3.2 Using Java RMI with PJama

This section examines the problems with using standard Java RMI *unchanged* in a persistent system. If standard RMI is used in a persistent context, remotely-invokable objects and the objects that hold references to them can become persistent by reachability from persistent roots. However, if these persistent objects are accessed in subsequent programs, they prove unusable because the data they contain describing the connection between them is as transient as the socket connection to which it refers. This problem is examined in more detail below.

If a client object, holding a reference to a remotely-invokable object in another VM, is

made persistent in one program, the client's reference will continue to work, in subsequent program runs over the same store. This will be the case as long as the server program that created the remotely-invokable object has continued to run in the meantime. The server-side thread continues to listen for socket connections and, at the RMI implementation level, the restarted client can use its existing information on connecting to the server to recreate the socket connection, the first time an RMI call is made to the server after restart.

Once the server program terminates, the next time the client object tries to use its reference, it will get a `java.rmi.ConnectException`, whether or not a program over the server store has been restarted before the client's latest call. The reason for this is that the socket connection to the remotely-invokable object is transient; it is associated with the specific execution of the VM that created it. Connection information is held in the stub at the client's VM. Except in the case of well-known services, the socket connection for a remotely-invokable object is likely to use different port numbers in different VM executions, but there is no facility for keeping the port number in a persistent client stub up-to-date.

Even if a remotely-invokable object is made persistent, PJama does not currently support persistent threads, so the thread that listens for incoming connections on behalf of the remotely-invokable object will be terminated when the program that created the remotely-invokable object is terminated. Attempting to re-activate the thread to listen for incoming calls after the server is restarted also does not work. If a server program attempts to do so by re-exporting the persistent, remotely-invokable object then, as a result of its call to the static method `UnicastRemoteObject.exportObject`, a `java.rmi.server.ExportException` will be raised with the message "object already exported". This is because exportation is necessary to create the listening thread but RMI does not support the re-entry of an object into the RMI implementation tables if it is already found to be there. Although the now-persistent object identity of the remotely-invokable object, as held in the client stubs and in the RMI implementation tables, could be used in a persistent context, there is however no existing support for making a remotely-invokable object with the same identity available across multiple program runs.

Applying the principle of orthogonal persistence to remotely-invokable objects and to the objects that use them remotely means that, ideally, their behaviour should be unaffected, whether or not they become persistent. To benefit from the resilience of a persistent client and/or server, PJama must incorporate additional support to ensure that remotely-invokable objects can be used remotely throughout their lifetime, and that objects holding references to them can use those references throughout their lifetime too. Supporting the illusion of continuous operation for such objects, throughout their lifetime, across multiple client and server program restarts, requires specialised support for maintaining the illusion of a persistent connection between the remotely-invokable object and its client. This is the support

provided by persistent RMI.

3.3 PJRMI: Remote Method Invocation Tailored for PJama

Having demonstrated that standard Java RMI will not work in a persistent context across multiple client and server program restarts, this section presents PJRMI: an enhanced implementation of RMI for PJama that is intended to solve the problems raised by combining persistence with distribution. This section focuses on the support provided by the release version of PJRMI for PJama running on JDK1.2 [PJR99], unless otherwise stated.

3.3.1 Persistent, Remotely-invokable Objects

Currently, PJRMI takes a conservative approach to the persistence of remotely-invokable objects; *all* such objects created in a PJama Virtual Machine (PJVM) running over a persistent store are automatically made persistent. This is intended to be a short term decision, on the basis that it is better to keep unused remotely-invokable objects in a persistent store, rather than to garbage-collect a remotely-invokable object mistakenly. Although this conservative solution is not scalable and uses up system resources unnecessarily, it is safe and it does support experimentation with persistent RMI. In the long-term, if we extend the notion of persistence by reachability across a distributed system then, given a reliable way of determining persistence by reachability from existing objects in other VMs, automatic persistence of all remotely-invokable objects would no longer be necessary. The difficulties of determining persistence by reachability across a distributed system are explored in section 6.

To address the problem of being able to use a persistent, remotely-invokable object, that relies on state external to the persistent system, beyond the duration of the program execution that created it, PJRMI uses the PJama mechanism called a `PJActionHandler`, as introduced in section 2.3.

The support enabled by `PJActionHandlers` is intended to recreate the transient state associated with remotely-invokable objects whenever necessary to ensure these objects continue to be usable as long as they are persistent.

In the first implementation of PJRMI[2], `PJActionHandlers` were used to re-export *all* persistent, remotely-invokable objects on every store restart. This ensured that every persistent, remotely-invokable object was available whenever the store was active. However, if a store was opened to support the use of one of these objects, all the others in that store were also

[2]Available in releases of PJama made during 1998: from version 0.4.6.12 to version 0.5.7.13.

re-exported, even though they were never used during that store run.

PJRMI now only re-exports each remotely-invokable, persistent object on its *first use* after store restart. This avoids unnecessary transfer of objects between persistent storage and main memory (object-faulting) and unnecessary use of system resources for objects not used in the current program execution; while still ensuring that the objects are available when they are required. Instead of a `PJActionHandler` instance being registered with the `PJActionManager` for each and every remotely-invokable object in the store, there is one `PJActionHandler` instance registered for all of them. This `PJActionHandler` re-exports one PJRMI-implementation-level, remotely-invokable object on every store restart, which acts as a well-known service, called `PJExported`. This service handles PJRMI-implementation-level requests from other PJVMs, to trigger the re-exportation of the specified object if it is not already available for use. Use of the `PJExported` service is described in more detail below.

3.3.2 Persistent Clients of Remotely-invokable Objects

PJRMI tries to maintain the illusion of a persistent connection between client and server by *automatically* re-establishing their connection on first use after store restart. Whereas the usage of transient sockets and threads makes it impossible to maintain this illusion for standard Java RMI, the PJRMI implementation ensures that if a PJVM is running over the store containing the required remotely-invokable object, the client will be able to use that object, *even* if the server PJVM has been stopped and restarted. If the server PJVM is not running, an exception is raised at the client to let it know that the referenced, remotely-invokable object is not currently accessible.

The client-side stub object is put into a state on store restart that indicates to PJRMI that, on first use, the service PJVM should be contacted to obtain up-to-date connection information for the stub. Using the `org.opj.utilities.PJSystem` method `markTransient` introduced in section 2.3, the connection information field of the stub class is marked transient; thus, PJama sets the field to `null` on store restart. Then, when the client object tries to make an RMI call via this stub, after client store restart, the PJRMI implementation detects the `null` connection field of the stub. It contacts the `PJExported` service in the PJVM running over the referenced, remotely-invokable object's store. Up-to-date connection information is obtained from `PJExported` and renewed in the client's stub object, allowing RMI calls to be resumed.

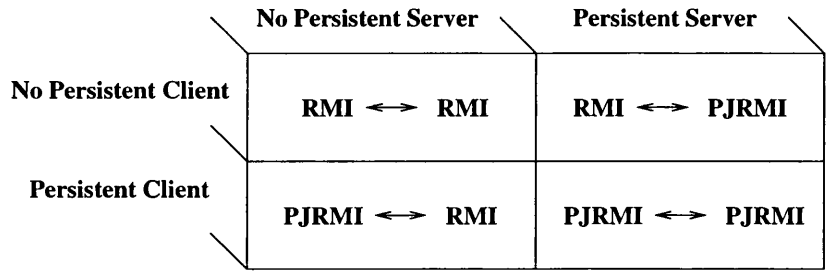

Figure 3.2: Permutations for communicating VMs in an open persistent system

3.3.3 Interoperability of PJRMI with Standard RMI

Given the open nature of the PJama persistent system, RMI communication between persistent and non-persistent VMs can potentially take on any of the four permutations illustrated by the matrix in figure 3.2. Some users of PJama use RMI for communication between a persistent server and transient clients: the server program runs on a PJVM and the client programs or applets run on standard JVMs. This raises the issue of compatibility between the versions of the RMI classes used by the JVM and those modified for PJRMI. Without support for class evolution, a standard JDK client would be prevented from communicating successfully with a PJama server using RMI, because of the mismatch between versions of the same class in the different VM implementations involved. PJRMI uses the class versioning support provided with Java Object Serialisation (JOS) [JOS98]. Changes made to RMI classes for PJRMI are compatible with standard RMI versions of those classes, according to the guidelines described in chapter five on the "Versioning of Serializable Objects" in the Object Serialisation documentation [JOS98]. The compatibility of the evolved class with the original is indicated by the inclusion in the evolved class of a field known as a `serialVersionUID`; the field contains a fingerprint of the original class, generated using a standard JDK tool. With such support, it is possible for two VMs holding different versions of the same class to communicate objects of that class between them successfully.

The class versioning support in JOS is minimal, when considered for use in a persistent system, and it is tailored to object serialisation. The class versioning support required for PJama is somewhat different. Given that the lifetime of a persistent store may be counted in years, the potential for changes to classes over time is high. Thus, more sophisticated support for class evolution is being investigated as part of the PJama project [Dmi98].

3.3.4 PJRMI Summary

The current implementation of persistent RMI for PJama (PJRMI) supports:

- the execution of standard RMI programs by a PJVM that is not running over a persistent store;
- the running of persistent RMI programs by a PJVM over a persistent store; where the latter includes support for:
  - persistence of all remotely-invokable objects,
  - persistence by reachability of objects holding references to remotely-invokable objects from remote VMs,
  - automatic re-exportation of persistent, remotely-invokable objects on first remote use after store restart and
  - automatic re-establishment of the connection between a remotely-invokable object and the object in another VM holding a reference to it, on first use of the reference after store restart;
- the compatibility of PJRMI with standard RMI to support RMI communication between a standard JDK VM and a PJVM.

PJRMI has been distributed with releases of PJama since April 1998. It has had a number of users outside of the PJama project whose feedback seems to indicate that this technology is usable and reliable. For more information on PJRMI users and their feedback, see section 3.5.

3.4 PJRMI Implementation Details

3.4.1 Using PJActionHandlers

As described in section 2.3, support is provided in PJama for associating callbacks, known as action handlers, with classes or class instances, to be run principally before stabilisation or on store restart. This allows the application programmer to set, re-constitute or tidy up the state of objects which may be viewed as persistent by a persistent application but which are actually objects external to the persistent system. These action handlers can be used to, for example, open and close sockets and files, open and close windows and, as long as there is no support in PJama for persistent threads, re-create and stop threads associated with a store.

A significant proportion of `PJActionHandlers` in the PJama platform are used for doing PJRMI-related actions. PJRMI associates `PJActionHandlers` with certain RMI classes for two purposes:

1. to re-initialise static fields of persistent objects and
2. to recreate intrinsically transient objects which cannot be made persistent.

Classes typically use static code blocks to initialise their static fields; this code is run when a class is first loaded into a JVM. However, once classes have become persistent, it is necessary to implement `PJActionHandlers` to rerun such initialisation code where required, before the class is used for the first time after a store restart. `PJActionHandlers` used to re-initialise the static variables of classes on store restart should ideally only be run on the loading of the appropriate classes from the persistent store. Running this reinitialisation code on each store restart brings every one of the classes registered for this reinitialisation into memory, even though the classes themselves may never actually be used during the current program execution over the store. There is a tradeoff between:

- paying the cost of running `PJActionHandler` code for a class during store restart that may prove unnecessary because the class is unused during the subsequent program execution and
- paying the cost of a check every time a class is loaded into the VM to see whether `PJActionHandler` code should be run before using it.

The former ultimately seems much less of a penalty, given that in PJama 0.5.20.2 for example, `PJActionHandlers` are associated with only twenty classes, which is likely to be a small proportion of the number of classes used in most persistent program executions.

The PJRMI implementation describes, in more detail, the use of `PJActionHandlers` where they are directly relevant to the implementation of PJRMI functionality.

3.4.2 Supporting Persistent, Remotely-invokable Objects

An object is made available for remote use (exported) either on creation, because it extends the class `java.rmi.server.UnicastRemoteObject`, or by making an explicit call to that class's method:

```
public static RemoteStub exportObject(Remote obj)
```

An addition to the code of class `sun.rmi.server.UnicastServerRef` for PJRMI ensures that every object exported in one of these two ways will be persistently-usable if the current VM is running over a persistent store. It does this with a call to the `saveIfPersistent` method of the class `org.opj.distribution.PJamaPJExported`:

```
public static void saveIfPersistent(ObjID id, Object o, RemoteStub s)
```

This enables the `PJamaPJExported` class to maintain a mapping between a stub's object

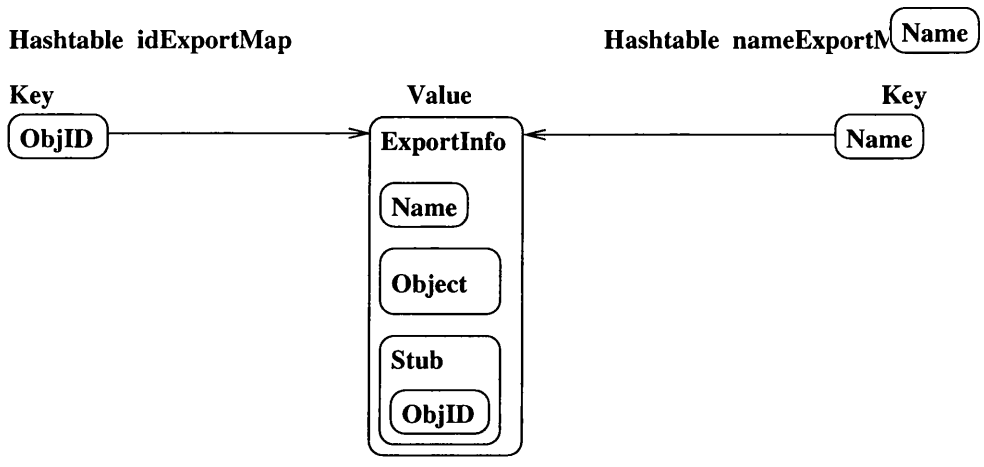

Figure 3.3: PJamaPJExported tables track export information by name and identity

identity and its corresponding remotely-invokable object.

A mapping is also created in the `PJamaPJExported` class for every object registered by name with the RMI Registry. Thus, given either a name or an object identity, the class `PJamaPJExported` has sufficient information to, if necessary, update and then return the connection information for the corresponding remotely-invokable object. The two tracking tables are illustrated in figure 3.3.

3.4.3 PJRMI Re-initialisation on Store Restart

The `PJActionHandlers` associated with PJRMI classes are principally used for re-initialisation of state on store restart. In summary, `PJActionHandlers` are run on all of the following RMI implementation level classes on every store restart:

```
org.opj.distribution.PJamaPJExported
sun.rmi.transport.DGCImpl
sun.rmi.transport.DGCAckHandler
sun.rmi.transport.DGCClient
java.dgc.VMID
sun.rmi.transport.ObjectTable
sun.rmi.transport.tcp.TCPEndpoint
sun.rmi.transport.tcp.TCPTransport
```

Their use is described in more detail below.

The code of `PJamaPJExported`, called during the exportation of the first remotely-invokable object over a store, ensures the persistence of an instance of the class `PJamaPJExported` too.

A `PJActionHandler` is registered when this object is created, to ensure that it is re-exported on every store restart. This makes the services of its `org.opj.distribution.PJExported` interface available whenever the store hosting the service is active. The uses of this service will become apparent during the explanation of automatic re-exportation for application objects below.

Distributed Garbage Collection (DGC) objects are used in the Java RMI implementation for tracking references between JVMs. They were designed to work for the lifetime of one VM execution. It is unlikely this implementation would be sufficiently maintainable or scalable for use over the lifetime of a store. Thus, although the state of DGC implementation objects can become persistent once RMI objects are in use over a persistent store, DGC tracking information is only valid within one program execution over a store. Thus, a persistent `sun.rmi.transport.DGCImpl` instance will also be re-exported on every store restart, to track any remote references created or recreated during the current program execution. On store restart, `PJActionHandlers` also re-initialise the static fields of the classes `sun.rmi.transport.DGCAckHandler` and `sun.rmi.transport.DGCClient`, to recreate their transient values.

On each store restart, the local IP address held in a static field of the `java.dgc.VMID` class, is reinitialised with a `PJActionHandler`. This demonstrates the need to reinitialise location-specific information associated with a particular VM execution since, for example, one program may be executed over a store on one host, while the next program may be executed over the same store but on a different host with a different IP address.

The class `sun.rmi.transport.ObjectTable` is used for maintaining the mapping from `ObjID` to remotely-invokable implementation object for servicing method invocations from remote sites. A `PJActionHandler` has been added to this class to ensure the state of its static tables is reinitialised on every store restart. Clean tables on store restart ensure successful re-exportation of persistent, remotely-invokable objects.

Other static connection-related tables are reinitialised on store restart. The `localEndpoints` table of the class `sun.rmi.transport.tcp.TCPEndpoint` and the table mapping threads to socket connections in the class `sun.rmi.transport.tcp.TCPTransport` are both recreated, since the information held in them from previous executions will be invalid for the current program execution.

Once the initialisation of store restart is complete, the application code for this run is invoked. At this stage, although a couple of implementation-level PJRMI objects are now actively available for remote use, the default support for application-level remotely-invokable objects is to leave them quiescent in the store until they are required. This ensures system resources are not taken up unnecessarily.

3.4.4 Supporting Persistent References to Remotely-invokable Objects

Extra support has been added to PJRMI to detect a client's first use after store restart of a reference to a remotely-invokable object. This section describes how this first use of the reference is caught and used, if necessary, to trigger re-exportation of the corresponding remotely-invokable object.

3.4.4.1 Obtaining a reference to a remote object

A client obtains a reference to a remotely-invokable object, either by looking it up by name in the RMI Registry or as the result of an RMI call on another remotely-invokable object. What the client actually gets is a reference to an instance of the stub class, derived from the interface supported by the remotely-invokable object and extending the class `java.rmi.server.RemoteStub`.

3.4.4.2 Preparing a Stub to Trigger Re-exportation

Every instance of `RemoteStub` that is passed to a client contains a `ref` field inherited from its superclass `java.rmi.server.RemoteObject`. This `ref` field contains the information necessary to create a connection from the stub back to the remotely-invokable object it represents, whenever the client uses it to make an RMI call. Since such connection information is only valid as long as the server process that generated it continues to run, static code has been added to the `java.rmi.server.RemoteObject` class to mark its `ref` field as transient, using the `markTransient` method of `org.opj.utilities.PJSystem`. This ensures that the `ref` field of a persistent stub is `null` after a store restart, when the connection information will probably no longer be valid.

The connection information in a standard RMI stub directs remote method invocations to the correct JVM location, while the object identity, an instance of the class `ObjID`, indicates which object at that location should service them. The `RemoteStub` class has its own code for serialisation and deserialisation of its instances, defined as `writeObject` and `readObject` methods with the signatures expected by Java Object Serialisation. On reception of a `RemoteStub` instance at its destination, its `readObject` method takes care of deserialisation of its connection information. For PJRMI, code has been added to this method to extract the `host` and `ObjID` from the `ref` and store it in fields of the `RemoteStub` itself. This ensures the information is available in a persistent stub after the `ref` itself has been set to `null`. Another extra field added to the `RemoteStub` class for PJRMI stores the `reexportPort` which, along with the existing `host` field, comprises the information necessary to make a connection to the `PJExported` service in order to update the stub's `ref` field

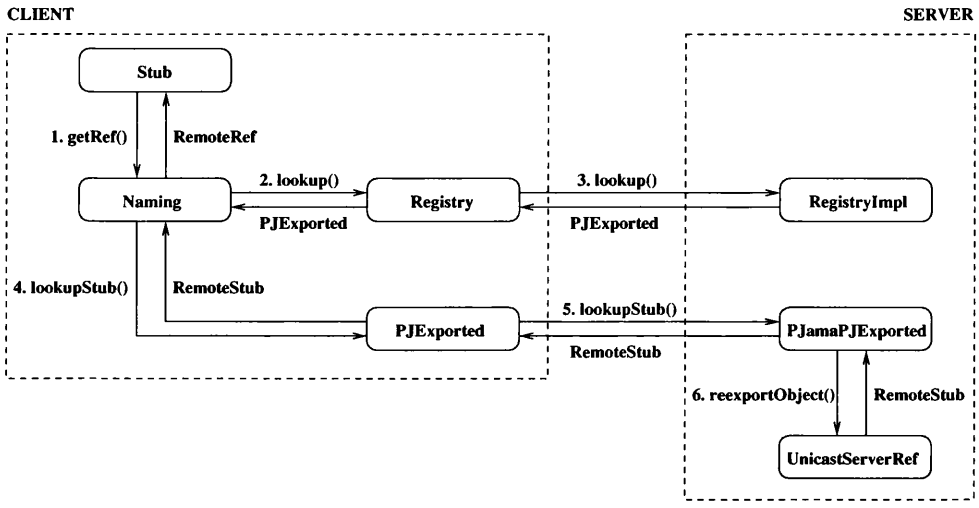

Figure 3.4: Renewing stub information

when it is found to be `null`.

3.4.4.3 Re-exportation on First Access

Once a stub has become persistent by reachability, the PJRMI implementation will detect first use of that stub after restart because of its null `ref` field and renew the stub's connection information. The method calls invoked to achieve this are illustrated in figure 3.4; they are numbered for ease of reference in the description below. The class `sun.rmi.Generator` is used by the Java RMI Stub Compiler `rmic` to generate the code for a stub from its corresponding remotely-invokable object class. The `Generator` class has been extended for PJRMI. Extra code is generated at the beginning of each stub method. It ensures that, when the stub's `ref` field is found to be `null`, the `getRef` method of the class `java.rmi.Naming` is invoked to renew the stub's connection information (method one in figure 3.4).

The `Naming.getRef` method first looks up the `PJExported` service at the `host` and `reexportPort` given by the stub (method two calls method three in figure 3.4):

```
PJExported pjexported = (PJExported) Naming.lookup(
  "rmi://" + host + ":" + reexportPort + "/PJExported");
```

It then uses the returned `PJExported` remote reference to make an RMI call to retrieve up-to-date stub information for the given `id` (method four in figure 3.4):

```
RemoteStub stub = pjexported.lookupStub(id);
```

The `PJamaPJExported` implementation of this call looks up the object with the given identity and re-exports it if it is not currently available for remote use (method five in figure 3.4).

The following method of `UnicastServerRef` is called to return an updated stub after re-exportation of the remotely-invokable `impl` (method six in figure 3.4):

```
public RemoteStub reexportObject(Remote impl, Object portData)
```

Note that during re-exportation the object retains its association with the object identity allocated to it during its initial exportation. This object identity is thus maintained across multiple program invocations. The client-side stub is refreshed with up-to-date connection information. The RMI call made on the stub, that triggered this re-exportation in the first place, can then go ahead as normal.

3.4.5 Interoperability of RMI and PJRMI

The original development of PJRMI focussed on client-server programs communicating using RMI where both client and server used PJama. This meant that both client and server picked up the same version of PJRMI classes, so that versioning was not an issue.

However, the first users of the release version of PJRMI were interested in using RMI for client-server communication where, although the server used PJama, the clients used a *standard* JVM. These clients, whether Java programs or Java applets, used the standard JDK RMI classes to communicate with a PJRMI service supported by the server. Failures during serialisation and deserialisation for RMI calls occurred, because the RMI classes at the clients were not the same version as the RMI classes modified to include PJRMI functionality at the server.

Responding to the users' feedback, a new version of PJRMI was released that exploits the versioning support that exists for Java Object Serialisation[3]. The issues raised and changes made to PJRMI to ensure support for all the permutations for communicating PJama and standard Java VMs, as illustrated by the matrix in figure 3.2, are described in the rest of this section.

3.4.5.1 Evolving Interfaces: the Effects on Stubs and Skeletons

When two VMs are participating in an RMI call, the class of the stub in the client VM must support exactly the same interface as the class of the corresponding skeleton in the server VM. To ensure this, a check is made in the code of every method of the skeleton (as described below), before it will forward a call from a stub to its remotely-invokable object.

The implication of this is that if an interface to a remotely-invokable object is evolved then

[3]For a description of the more sophisticated class evolution support now being provided with PJama, see [Dmi98].

a client, that obtained a reference to the object before evolution took place, will no longer be able to use it. This applies even to theoretically-acceptable forms of evolution, such as only adding new methods to an existing interface, while continuing to support the old ones. The client would need to be able to replace the old version of the stub class with a newly-loaded one and obtain an instance of the new stub class before being able to resume use of services provided by the evolved RMI service interface. This would be a challenging task in standard Java. Since the type equivalence of classes in Java is based on their name and classloader, replacement of one version of a class with another version of the same class is non-trivial using Java alone. Given the potential for long-lived classes in a persistent system, there is a need to address this issue though. Thus, an off-line tool is provided with PJama, called `opjcs`, which does support the substitution of one version of a class with another version of the same class in a persistent store, as described in [Dmi98].

The check for a matching interface at client and server is implemented in standard RMI as follows. At compile time, the `rmic` compiler sets a private static field to the same value in both the skeleton and stub class; this field contains a hashcode generated from the signatures of each method of the corresponding remotely-invokable interface. Thus, for the standard JDK `Registry` interface, calling `rmic sun.rmi.registry.RegistryImpl` generates the `sun.rmi.registry.RegistryImpl_Stub` and `sun.rmi.registry.RegistryImpl_Skel` classes, both of which contain the field:

```
private static final long interfaceHash = 4905912898345647071L;
```

At run-time, when a remote method invocation is made from the client, the stub forwards this call to the server, including the `interfaceHash` field of the stub class as a parameter. At the server, the skeleton checks whether the given `interfaceHash` from the client matches the `interfaceHash` field of its own class, before making the method invocation on its intended target. If the `interfaceHash` fields of stub and skeleton do not match, a `java.rmi.server.SkeletonMismatchException` is raised.

The PJRMI implementation originally included the addition of an extra method to the interface `java.rmi.registry.Registry`, but this caused a mismatch of the PJRMI Registry interface with the standard JDK Registry interface. It was possible to revise PJRMI so that this additional method could be removed, leaving both PJRMI and RMI with the same `Registry` interface once more.

The implication of this restriction on the evolution of interfaces is that when client references to remotely-invokable objects are made persistent, long term maintenance in the face of an evolving system is difficult without sophisticated evolution support [Dmi98].

3.4.5.2 Evolving Classes to Handle Multiple Versions

Unlike interfaces, there is support for having Java classes at different versions in the two VMs involved in an RMI call. The Java support for serialising instances of classes does take compatibility of the different class versions at source and destination into account.

This support comes in two parts. Firstly, object serialisation code must be written for the evolved class to handle serialisation and deserialisation of objects created with the original version of the class, as well as the evolved one. Secondly, a field must be added to the evolved class to indicate that it is now compatible with the original.

Adaptable Serialisation

Standard Java Object Serialisation includes support for serialising and deserialising different versions of the same class which works quite well; as long as the programmer respects the recommendations of the JOS documentation [JOS98] meticulously.

Where evolution of a class involves the addition of new fields which are to be serialised, the programmer must create or extend `writeObject` and `readObject` methods for the evolved class, to handle serialisation and deserialisation correctly. They must also ensure that they do not perturb the writing and reading of the original class when extending the code for the evolved class.

When the default serialisation provided by `java.io.ObjectOutputStream` applies to the original class, it can handle the automatic serialisation of the additional fields of the evolved class too. However, if a `writeObject` method exists in the original class, which writes out fields explicitly, it may be necessary to extend it to ensure the additional fields are serialised.

Where the default deserialisation provided by `java.io.ObjectInputStream` applies to the original class, this is not sufficient for an evolved class with additional fields. If an original class version is expected but an evolved class instance is supplied, the extra fields of the evolved class will automatically be skipped. However, if an evolved class version is expected but an original class instance is supplied, the default serialisation code will expect to deserialise more fields than the stream contains. Thus, a `readObject` method for the evolved class must be created or extended to handle deserialisation of instances of both the original and the evolved class. For the `readObject` method of the evolved class to determine whether it is currently deserialising an original or evolved instance of the class, it must use the `java.io.ObjectInputStream` method:

```
public int available()
```

to determine whether any more bytes are available, before trying to read the extra fields of the evolved class.

Adapting Serialisation for PJRMI Classes

In order to make certain PJRMI classes compatible with their standard JDK originals, it was necessary to make some modifications to their serialisation code. The standard JDK version of the stub class `java.rmi.server.RemoteStub` contains no `writeObject` or `readObject` methods at all. Because of the addition of extra fields to the PJRMI version of the class `java.rmi.server.RemoteStub`, `writeObject` and `readObject` methods were added to the evolved class. These new methods contain calls to the original default serialisation code, before the code for serialisation and deserialisation of the extra fields, to ensure the original serialisation is still maintained correctly.

The code of the evolved class's `readObject` method uses a call to the method `available` of class `java.io.ObjectInputStream` to determine whether the extra field of the `RemoteStub` is in the stream before trying to read it. Where a standard JDK version of a `RemoteStub` is being read, this call would return zero.

Indicating Compatability

Java Object Serialisation relies on the use of fingerprints generated from a class to indicate compatibility of class versions. A `serialVersionUID` is generated from the original class and incorporated as a static, final field of the evolved class, whenever it is appropriate to indicate the compatibility of the evolved class with the original. Successful use of this across sites requires programmers to be diligent about incorporating `serialVersionUIDs` where appropriate into evolved classes.

The `serialVersionUID` is a fingerprint of the class, similar to the `interfaceHash` used for interfaces. It is generated from the method signatures of the class and the field names and types of every non-transient, non-static field of the class. This is done by running the Java executable `serialver` with the original of a class as its parameter, as illustrated by the example below:

```
susan@kona31:  serialver java.rmi.server.RemoteStub
  java.rmi.server.RemoteStub:
  static final long serialVersionUID = -1585587260594494182L;
```

Adding the resulting field

```
static final long serialVersionUID = -1585587260594494182L;
```

to the evolved class indicates its compatibility with the original.

The runtime check for class compatibility occurs during deserialisation. Every object, marshalled using Java Object Serialisation for RMI communication between two sites, is preceded by a class descriptor indicating the class name and fields of that object. Every class descriptor includes the `serialVersionUID` for that class. The site unmarshalling a se-

rialised object will only do so if the class descriptor's `serialVersionUID` matches the `serialVersionUID` of the class with the same name in the unmarshalling site's VM. The following message is an illustration of the exceptions raised when the `serialVersionUIDs` do not match.

```
java.rmi.UnmarshalException: Error unmarshaling return;
  java.io.InvalidClassException: java.rmi.server.RemoteStub;
     Local class not compatible:
       stream classdesc serialVersionUID=-5354926258777194346
       local class      serialVersionUID=-1585587260594494182
```

Where the `serialVersionUIDs` do match, this means that an object serialised at source with one version of the class can safely been deserialised using the other version of the class at its destination.

Indicating Compatability for PJRMI Classes

For PJRMI, it was necessary to generate and add `serialVersionUID` fields to the PJRMI version of the `java.rmi.server.RemoteStub` and `java.rmi.server.RemoteObject` classes, after ensuring their compatibility with their standard JDK originals.

3.4.6 Implementation Revisions

3.4.6.1 Automatic Stub Class Generation

In standard RMI, after compilation of a remotely-invokable object class, it is necessary to invoke a separate `rmic` compiler on this class to generate corresponding stub and skeleton classes. This must be done before the application code for creating a remotely-invokable object of that class can be run.

In the early releases of PJRMI, dynamic, automatic generation of the stub and skeleton classes was introduced. A call to the `rmic` compiler was added to the code for exporting an object. This was done because, from the programmer's point of view, it removes an extra and easily forgotten step for compiling code for remotely-invokable objects; thus also removing a common source of errors in running RMI programs without stub and skeleton classes being available. (The drawback with this approach is that, if the class definition causes an error during stub generation, this only becomes apparent at runtime.)

However, although the cost of calling the compiler at run-time is incurred only once per remotely-invokable object exportation, this cost can be noticeable to the user. Another problem is the question of where to put the automatically-generated classes. They cannot just be

created in-memory in the VM doing the exportation, since they must be available as class files to remote client VMs that need to pick up the stub class in order to be able to use its corresponding remotely-invokable object. Creating them in the current directory for the executing application proved confusing and too restrictive for PJRMI users; particularly when these classes had to be made available from a codebase for downloading by applets. It was not obvious that the user would have less problems explicitly stating where the class files should be written than they had with using `rmic` themselves.

Avoidance of the extra compilation step for remotely-invokable objects did not prove to be sufficiently warranted to cope with the problem of where these class files should be created, so the runtime generation of stub and skeleton class files was dropped when PJRMI was ported to PJama running on JDK1.2.

3.4.6.2 Use of the RMI Registry for PJRMI

The RMI Registry is provided as part of standard RMI. It is a well-known service, supporting look-up by name of remotely-invokable objects on the Registry's host machine. Clients can use the Registry to obtain a reference to an object in a remote JVM. This sort of service is often used for bootstrapping the interaction between two VMs.

It seemed reasonable, since the Registry is such a useful service presented as part of standard RMI, to recommend that a Registry be installed in every store that is to contain remotely-invokable objects. A `PJActionHandler` was written for the `RegistryImpl` class that supports the `Registry` interface, so that after a Registry is made persistent, it is then re-exported on every store restart.

Given the persistence of the Registry, it then seemed appropriate to add to it the PJRMI functionality for supporting persistent, remotely-invokable objects. It is necessary to track all persistent, remotely-invokable objects in order to support their re-exportation after store restarts. The first design for PJRMI identified the Registry as a suitable object to host the data for such tracking.

Thus, in the first pre-release version of PJRMI, the class `RegistryImpl` contains a hashtable supporting the lookup, by object identity, of information on an exported object. However, feedback from pre-release users indicated that they required more flexible use of the Registry than was recommended for PJRMI. The recommendation of a Registry per store was not popular with one user who wanted several stores on one host machine to share a single Registry. The mere existence of the Registry in the store was not popular with another user who wanted to implement their own lookup service as a replacement for the Registry in their application.

The user feedback prompted a review of the design. A more modular design was produced, separating the functionality of the Registry name service from that of the service tracking persistent, remotely-invokable objects. A new class `PJamaPJExported` was introduced to hold the information on the persistent, remotely-invokable objects. PJRMI now requires that an instance of this class exists in every store containing remotely-invokable objects instead. Since `PJamaPJExported` is purely used at the PJRMI implementation level, unlike the Registry, this avoids the clash with user requirements.

3.4.6.3 Re-exportation of all RMI objects on Store Restart

As described in section 3.3.1, in the first releases of PJRMI, all persistent remotelyinvokable objects were re-exported for remote use on every store restart. PJRMI now only re-exports each remotely-invokable, persistent object on its *first use* after store restart.

3.5 Using PJRMI

This section is intended to give the reader information on the impact of using PJRMI. Some recommendations on taking advantage of persistence are made for PJRMI in section 3.5.1. The section also references the PJRMI tutorial that is included in the dissertation as appendix A. Section 3.5.2 presents the experiences of real users, whose feedback has been beneficial in improving PJRMI, as described in section 3.5.3.

3.5.1 Model of Usage

Using PJRMI support, it is possible to take the code of remotely-invokable services and their clients, as written for standard RMI, and use them in a persistent context *unchanged.* Alternatively, when developing code from scratch, a recommended model of usage can be followed for PJRMI, that takes advantage of persistence.

In a standard RMI program, a service supported by a remotely-invokable object is created and exported for remote use and then, as the program continues execution, it waits to service incoming method calls from other JVMs. If the program execution is killed, then the next time that service is required, the program must be run again, creating and exporting the remotely-invokable object anew, so it can continue servicing RMI calls.

In the persistent RMI model, the remotely-invokable object can be created once, made persistent and is then available in the persistent store to service incoming method calls during future sessions that use that store. This works on the basis that typical persistent application usage involves populating a store once and then using the store contents repeatedly. Having

populated a store with persistent, remotely-invokable objects, subsequent programs running over that store will find that these objects are still available for remote use.

A tutorial, developed to introduce PJama users to PJRMI, illustrates the differences between writing a standard RMI program and taking advantage of persistence in a PJRMI program. A recent version of the tutorial, provided as part of the documentation for the PJama release version 0.5.20.2, is included in appendix A. The tutorial is fully illustrated with working code examples taken from the PJRMI demo programs, which are also included in each PJama release. Having introduced a standard Java RMI program in section A.2, section A.3 then builds on this example to show what changes are necessary to take advantage of persistence with PJRMI. Section A.4 presents an example of a program that can be used to cleanly shut down a persistent store containing remotely-invokable objects. Section A.5 concludes the tutorial with a list of common exceptions and their probable causes, to aid in diagnosis of problems that may occur during the execution of the example programs.

3.5.2 User Feedback

Extracting feedback from users and applying it to further iterations of the design process is an important part of the software development cycle. A number of different methods have been used to obtain feedback from users of PJRMI. A user is usually identified initially by the complaints they send to the author, about PJRMI not working or not being what the user wants or expects. Follow-up emails have been used to extract details of what the users are using PJRMI for and what they think of it. PJRMI has been used by a number of users since its inclusion in releases of PJama from April 1998 onwards. Some specimens of PJRMI users and their applications are presented below.

3.5.2.1 The DRASTIC Project, University of Glasgow, Scotland

A distributed system has been developed by the DRASTIC project at the University of Glasgow, for supporting the run-time evolution of classes and objects at run-time [ED97, ED99]. Originally developed in Modula-3, it was ported to Java and used object serialisation for persistence, before ultimately being ported to PJama to make use of orthogonal persistence. A distributed application supported by DRASTIC is divided between a number of zones, where each zone is a logical collection of processes. Zones are the unit of evolution within DRASTIC. A zone contract is defined between a pair of zones. The contract specifies the types that may be exchanged between the two zones and the transformations that need to be applied to any remote invocations or object migrations that take place across the zone boundary. Zones and contracts provide support to allow the software engineers to contain

and cope with the evolution of their classes and the whole system. Communication across the distributed system is done using RMI.

A paper written on "Porting a Distributed System to Persistent Java: An Experience Report" [ES98] identified a number of problems and made some comments on PJRMI.

Flexible use of the RMI `Registry` was identified as important for DRASTIC. Rather than being required to have one running on every machine hosting a remotely-invokable object, as required by standard RMI, the preference was for a single `Registry` for the whole system, for use by all hosts. This type of requirement did not result in a relaxation of the one-per-host requirement of the `Registry` for PJRMI, since that would be contrary to the standard RMI design, but it did imply both that the `Registry` was an unsuitable place to focus the PJRMI functionality on a per-VM basis and that there should be flexibility over the persistence of the `Registry` itself.

It was rightly pointed out that, given the orthogonal persistence of PJama, remotelyinvokable objects, like any others, should only become persistent if they become reachable, directly or indirectly, from an application object which has been registered as a root of persistence. Despite the documentation on this current *feature*, there was some initial confusion of expectations over what becomes persistent; however, the persistence of all remotely-invokable objects did not prove problematic for DRASTIC in practice.

PJRMI tries to retain referential integrity between a remotely-invokable object and the clients that reference it. This involves automatically updating connection information on persistent references at the clients in order to ensure a persistent connection to the remotely-invokable object. It is noted in [ES98] that referential integrity can never be guaranteed in a distributed system, due to the potential for network and host failure. A criticism of the automation was that application programmers have no control over how it happens or the ability to run application code immediately before or after the connection is re-established. It was commented that it would be useful for the application programmer to be able to switch off automated reconnection. However, PJRMI chooses to try, as far as possible, to maintain referential integrity and at least the illusion of persistence across the distributed system. It does this in order to try to provide orthogonal persistence across distributed VMs, as well as within one VM.

3.5.2.2 TuaMotu: The ECOO Project, LORIA, France

The ECOO (Environnements pour la COOpération) project [ECO00] is based at the French research institute LORIA (Laboratoire Lorrain de Recherche en Informatique et ses Applications) in Nancy, France. The researchers on this project have been working for some years on developing distributed support environments for cooperative work, with empha-

sis on the use of objects over wide-area networks. Their recent work on a system called "TuaMotu" [CBGM98, CMG98] has included an evaluation of PJama and other persistence technologies for provision of support for persistent object management services. The researchers Jean-Marc Humbert and Pascal Molli have provided feedback on their experiences working with PJama.

PJama feedback

The alternatives considered for persistence support include PJama, POET [POE98], Java Object Serialisation [JOS97], JOP (Java Object Persistence) [JOP96] and Enterprise Java Beans (EJB) [EJB99c]. The application was first implemented using Java Object Serialisation. However, PJama proved in comparison to be "the best one solution I've tested" so far. It was described as "a very flexible solution", because the memory management for persistent objects is integrated with the existing managed heap and garbage collection of Java, the writing of Java objects to persistent storage is done automatically and the persistence is orthogonal to type, removing any requirement to specify which types can persist.

The lack of changes to Java code to use persistence was seen as a major benefit of PJama. In comparison, the POET database required the adaption of TuaMotu package structures and did not support hashtables transparently, which was unfortunate since a lot of them are used in TuaMotu. The alternative implementation of hashtables provided by Poet does not support the same methods as `java.util.Hashtable`. Using POET with TuaMotu was stopped because of the number of things that had to be modified to get them working together.

JOP, which does perform automatic mapping of Java objects to a relational database using JDBC, has also been under trial, as has EJB.

PJRMI Feedback

For TuaMotu, PJRMI is used for client-server communication. The server hosts a single remotely-invokable object, modelled on the Command pattern[4], to represent the server application. Clients look up the server's RMI object in the RMI Registry and then send commands to the server by passing command objects as parameters in RMI calls:

```
server.send(cmd)
```

A command sent from a client to the server may include a reference back to a remotely-invokable object available at the client, to be used during execution of the command by the server to make callbacks for event management. A series of commands forms a transaction, terminated by the command `EndAct`. The server is designed as a global transaction with checkpointing; each time a client commits a short transaction to change the state of the server, a call is made at the server to ensure the changes persist in the store.

The application is designed to make a clear separation between volatile (short-lived) and

[4]For information on the Command pattern and other design patterns, see [GHJV95]

persistent objects. However, when the ECOO group tried using a version of PJama, even before the inclusion of PJRMI, it was found that the server's remotely-invokable object was made persistent by reachability, whereas this was not the case in the standard Java version of the application using Object Serialisation. This did not prove problematic, after they received a version of PJama incorporating support for PJRMI so that persistent, remotely-invokable objects would still be usable, but they did observe that a tool for inspecting the objects stored in a PJama store would be useful for users to confirm whether or not objects have been made persistent.

Support for persistence was only required at the server, for storing application data. It was important for the clients to run standard JDK code, not necessary at exactly the same version of the JDK as the server. The application's RMI communications between client and server soon revealed the incompatibilities of the first version of PJRMI with standard RMI. The problems with versioning, as described in section 3.4.5, were identified, fixed and distributed in a subsequent PJama release.

3.5.2.3 A Hierarchical Archive: University of Hamburg, Germany

Two students, Norbert Schuler and Michael Otto, in the Software Engineering Group of the Computing Science Department at the University of Hamburg, were set a project to use PJama to make a hierarchical, multi-user archive persistent [OS98]. After some initial confusion over the setting of environment variables (a common problem with Java), they had no problems getting their application working with PJama and PJRMI. However, they did have a few problems with how PJama and PJRMI fitted in with the design of their application.

They built their application on top of an existing framework called JWAM, an implementation derived from a theoretical model of software engineering called WAM, used at their university. This framework provides a service for easy communication between processes; internally it uses RMI. An application incorporates a capsule providing this communication service into its implementation.

Since the implementation of the capsule providing the communication service is multi-threaded, and since persistent threads are not currently supported in PJama, it was not possible to make this capsule persistent. However, since the RMI objects in the framework are created in this application over a persistent store, they are automatically persistent anyway. The students instead marked the service as transient and wrote code to recreate it after every store restart.

The framework classes use the Singleton design pattern, which ensures that only one instance of a class exists: a static field of a class is set to reference the single instance of that

class. In order to make the framework services transient, the static singleton field of each of these classes could either:

- be marked transient from the code that creates the store – which requires this store creation program to have internal knowledge of the framework classes; or
- be marked transient from static code added to the classes themselves – which requires modification of the framework classes for use with PJama.

Neither of these options demonstrate a clean separation between the framework for communication between processes and the support for persistence used by the application. In the end, the students went with the first solution, to avoid making any change to the code of the framework classes themselves.

Their concern for a clean separation of the support for persistence from other parts of the system also extended to the RMI Registry. Although the students successfully built a solution with PJRMI's integrated, persistent Registry, they really wanted to run it only as a separate, external and non-persistent process, as it is supported in the standard JDK. However, because of the reliance of the PJRMI service `PJExported` on the existence of a Registry in its store, it was not possible to have only a non-persistent Registry in the system.

Ultimately, the students observed that the combined support provided by PJama and PJRMI is "not quite optimal" yet. While orthogonal persistence should be safe and easy to use, they identified the following problems as the most important obstacles in the way of these goals: lack of support for persistent threads, having to mark fields transient *explicitly*, occasional crashes of PJama during stabilisation and a lack of integration of support for orthogonal persistence with releases of the standard JDK.

3.5.2.4 007 Benchmark Server: Australian National University, Australia

Two researchers, Steve Blackburn and David Walsh, at the Australian National University in Canberra, Australia, have been working in the Advanced Server Technologies program on the UPSIDE project (Utilising Persistence and Scalable Information management in Distributed Environments). This project involves designing scalable transactional object storage systems for use with orthogonally persistent systems and languages.

One of the applications they have worked on is composed of a PJama server providing support for querying a 007 database plus non-persistent clients. It was originally a port of the 007 benchmark from its C++/PSI version to a version for PJama; done by Luke Kirby, an honours student at ANU. Walsh then removed the timing code and added a control loop to accept query requests from a client.

At the server a store is created, the standard PJRMI support services `Registry` and `SuspendService` are added, the 007 service is created using `007.server.CreateService`, the store is populated with the 007 tiny database using `007.server.GenDB` with parameter `oo7.config.tiny` and the service providing access to this database is run by invoking `007.server.Server` with parameter `oo7.config.tiny`. Non-persistent clients can then be run to trigger a range of queries over the 007 database at the server.

Walsh reported that he did find PJRMI easy and intuitive to use. He commented that "it would have been difficult without your supporting documentation. This explained the RMI differences quite well." Since the application used the model of a persistent server with standard JDK clients, like the researchers at LORIA, they initially came across the same problems with incompatibility of class versions between standard RMI and PJRMI, that existed in the early PJRMI releases. With the solutions to these problems provided in a subsequent PJama release, they did get their software working successfully, using PJama and PJRMI.

3.5.2.5 Distribution, Object-orientation and Persistence: University of Adelaide, Australia

Kevin Lew Kew Lin, a PhD student supervised by Fred Brown at the University of Adelaide in Australia, wrote his thesis on "orthogonal persistence, object-orientation and distribution" [Lin99]. A description of the work is included below.

"This project is investigating techniques to extend the benefits of the persistence abstraction to wide area networks where distribution must be explicit and network failures and delays are a significant programmer concern. Contributions of this project will include a locality mechanism, a network wide indirection mechanism and a model for distributed programming over confederated persistent object stores. Confederated stores exhibit the property of autonomous control with limited interactions with other stores. An indirection mechanism is to be provided to identify and address those services that stores wish to publish. Localities are an essential modelling mechanism to control pointer leaks and allow programmers to reason about store interactions that do not permit pointers between stores. "

Lew Kew Lin built a structure of logically nested "localities" i.e. nested persistent stores, implemented as a tree of directories containing PJama stores. PJRMI is used for communication within an application distributed over these nested localities and over distributed stores. He developed an "indirections" mechanism, which supports the dynamic registration of arbitrary objects as network services, without the need for stubs or precompilation, and light-weight calling of the services. Experiments with some simple applications compared the performance of this indirection mechanism with that of RMI and PJRMI.

For this work, Lew Kew Lin was the first external user of PJRMI after it was ported from PJama on JDK1.1.x to PJama on JDK 1.2 FCS (PJama version 0.5.20.0). Kevin was one of the few users who saw the persistence of the RMI Registry for PJRMI as a good thing. He also made a valid criticism of the tutorial example program for making a client persistent, saying: "I found it a bit confusing to have to

1. first write a message client that holds onto a remote reference, then
2. put the message client in the store and then
3. write programs that access and use the message client.

After understanding what was happening, I found it simpler to directly put the remote reference as a persistent root in the store and then write programs to access and use it." This demonstrates the importance of making example programs as direct and simple a demonstration of the technology as possible.

Since the `support.service.persistent.CreateSupportServices` PJRMI example program is only ever called once to prepare a store for remote interaction, he also suggested that it could in fact be an automatic step in store creation. This program creates two support services and makes them persistent: the RMI Registry and a `SuspendService`. The latter supports a remote call to shut down an otherwise indefinitely-running server cleanly. However, the feedback from other users indicates that automatically making the `Registry` persistent in every store is not a popular choice. Also, not all stores contain any remotely-invokable objects at all, so adding services automatically for their support is not necessarily helpful.

3.5.2.6 The Distributed Bibliography System: University of Glasgow, Scotland

Irene de las Heras, a Spanish ERASMUS student, worked on a project using PJama and PJRMI. She successfully developed a Distributed Bibliography System. The bibliographic server runs over a persistent store containing a collection of bibliographic entries. It services queries and performs updates on the collection. The client can be run as either a persistent application or as a non-persistent application or applet; it communicates with the server using RMI. It is invoked by users to, for example, request one or more bibliographic entries from the server based on given search criteria. Users can also add new bibliographic entries to the existing collection and create their own views of the entries. The information on users is maintained in the server's persistent store, including each user's views of bibliographic entries and each user's sets of entries.

Though Irene's report was lacking in evaluation of the PJama and PJRMI technologies that she used, she appeared to have very few problems with using the technology and developed a reliable distributed application with a good interface.

3.5.2.7 Bioinformatics project: University of Glasgow, Scotland

Iain Darroch is a researcher working on a Bioinformatics project. For this project, a PJama store has been populated with genetic map data. It is accessed by an applet that supports the displaying of a genetic map. To support RMI calls between the applet and the store, the store is hosted by the same machine as the WWW server from which the applet is downloaded.

Iain found it "reasonably straightforward" to follow the PJRMI tutorial. However, he did comment that the design of the tutorial examples (like the JavaSoft standard RMI examples) does not scale to a large application. He would have liked some guidance on design techniques to use at the larger scale, such as the adaptive design pattern [Bec99, GHJV95].

He found it difficult to diagnose the cause of RMI errors in his program. The RMI mailing list helped solve most of the problems he encountered. Like many others using RMI for the first time, he initially had problems with looking up the correct service on the correct host. He also had evolution problems: although he successfully used the PJama evolution tool `opjsubst` to substitute the original version of a service in the store with a new version containing an extra method, the tool couldn't pick up the implicit dependency between the service implementation class and the corresponding stub and skeleton classes in the store. Thus, the interface supported by the stub no longer matched that supported by the service implementation class and this resulted in errors on the next lookup of that service.

3.5.3 The Effects of Feedback

Feedback from users has influenced PJRMI development: descriptions of some resulting changes have already been indicated in section 3.4 on the PJRMI implementation; a summary of changes and observations is included below:

- Early feedback from Huw Evans (section 3.5.2.1) helped to motivate the separation of persistence support from the implementation of the `Registry`, moving it to its own class `PJamaPJExported` instead.

- Both the users at Loria (section 3.5.2.2) and ANU (section 3.5.2.4) required that PJama be used as a server and PJRMI used to communicate with clients which were either standard Java applets or standard Java programs. This raised the issue of interoperability between PJRMI and standard RMI, resulting in the changes described in

section 3.4.5.

- The students in Hamburg (section 3.5.2.3) wanted to encapsulate the persistence mechanisms used in their software to fit with the internal framework structure of their system, in order to follow good software engineering practices. However, the registry could not be encapsulated because of its use by remote components but is required to be persistent by the current implementation of PJRMI. The PJRMI requirement for the persistence of the `Registry` should be reviewed in the future.
- Users of PJama do have some idea of which objects they expect to become persistent by reachability and which should not. It has been demonstrated by some users' feedback that the persistence of *all* remotely-invokable objects does sometimes conflict with application design and users' expectations. Separation of persistence from other concerns of the system can be compromised, even though, due to the support provided by PJRMI, applications with persistent, remotely-invokable objects do work. A cleaner separation of persistence and distribution support is still a goal for PJRMI.

Feedback from users will continue to influence future work on PJRMI.

3.6 PJRMI: Could Do Better

PJRMI adds support to RMI for persistent, remotely-invokable objects and persistent references to them. It solves the problems created when non-persistence-aware RMI objects are pulled into a store through persistence by reachability. This includes maintaining the illusion of a persistent connection between a remotely-invokable object and a persistent reference to it. However, it also illustrates some problems with combining persistence and distribution.

Clients can obtain, and make persistent, references to remotely-invokable objects, either via a Registry lookup by name on a registered remotely-invokable object or by obtaining a reference to one from another object. Over time, it is possible for many references to be built up between distributed, persistent stores, creating dependencies between them.

RMI passes objects as parameters to remote method calls and as return values from them. If the object inherits from the interface `Remote`, it will be passed by reference. If it doesn't inherit from Remote but does inherit from the interface `Serializable`, the whole transitive closure of the object graph will be passed by *copy*. PJRMI may end up serialising and sending very large object graphs from one persistent store to another, including graphs that have built up incrementally in the persistent store over many program executions.

The rest of this dissertation presents the approaches taken for PJama to address these problems.

Chapter 4

Approaches of Related Work

4.1 Introduction

Experience with PJRMI has identified problems with orthogonal persistence when it is used in a distributed system. This section briefly sets the context for this dissertation and reiterates the problems themselves. The rest of the chapter examines the approaches of related work. Existing systems with the same potential problems are identified and the extent to which they have dealt with them is evaluated.

4.1.1 Context

Much work has been done on persistent systems over the last twenty years or so. A number of significant contributions in this field are referenced below to give the reader some context for the work examined in this dissertation.

Persistent systems have been developed based on a variety of languages. Examples include Pascal/R [Sch77], the E programming language [RC89] and Texas [SKW92], which are both based on C++, and the Mneme persisent object store [Mos90b], versions of which have been used with Smalltalk [HMB90], C++ and Modula3. Systems that respect the principles of *orthogonal* persistence presented in [AM95] include PS-algol [ABC+83], P-Pascal [Ber91] and Napier88 [MBC+96].

Most persistent systems are developed as virtual machines to run on top of conventional operating systems, but other approaches have been also been taken. Grasshopper [DdBF+94] is an example of an operating system designed to support orthogonal persistence directly, for greater efficiency. PSI, a Persistent Store Interface [Bla98], was developed as a result of investigations into the scalability of orthogonally-persistent systems, in the context of a

multi-computer architecture.

Both Grasshopper and PSI are also concerned with issues of distribution. Other systems concerned with these issues include those mentioned below. The Argus object-oriented programming language [Lis88] was developed specifically for distributed programming, with support for atomic objects to ensure the consistency of the persistent data used. A model of distributed programming was integrated with PS-algol [Wai88]. The addressing of large, distributed collections of persistent objects was examined in the context of Mneme [Mos90a]. A persistent RPC was implemented for Napier88 [dSAB96]. The work of this dissertation builds on that of these and other systems, focussing specifically on the two problems described below.

4.1.2 Problem One: Maintaining Object References Between Stores

Use of orthogonal persistence in a distributed system implies that it should be possible to make references to remote objects persistent. Support for persistent references to remote objects requires persistence of the reference itself, the subsequent persistence of the referenced, remote object and the continued persistence of the remote object, as long as the reference to it persists.

The persistent of references between stores is useful for increased simplicity and reliability of application execution in the short term, but such dependencies between stores threaten the maintainability of the stores involved in the long term. The store providing a service may be obliged to provide remote access to objects for as long as references are held to them from other stores. The store holding a client reference to a remote object is dependent on the remote store for its own referential integrity.

Where existing work provides support for references to remote objects, the manner in which those references are managed is considered. Where references can persist, the handling of the implications of such support is examined.

4.1.3 Problem Two: Copying Large Object Graphs Between Stores

Large graphs of objects can be created and made persistent or grown incrementally in a persistent store. With machines that can now have gigabytes of RAM, the size of in-memory object graphs created and made persistent by an application can correspondingly become large relatively quickly; easily megabytes in size.

Given the potential for large object graphs, of at least megabytes in size, some management for the copying of persistent object graphs between stores is necessary. However, since policies for the passing of objects across a distributed system are typically defined statically,

there is a lack of flexibility for adapting the management of copying of persistent object graphs between processes to cope with their size.

Once an object is persistent, if the object passing policy is static and thus persistent too, the resulting lack of flexibility has other implications too. It affects copying in the face of changes in use of a persistent object graph by different applications, since these applications may each have their own differing object passing requirements. Changes in distributed environment can also affect the handling of copying during the object graph's lifetime.

Existing systems, with support for copying object graphs between distributed sites, are examined in detail in section 4.2 for their approach to dealing with these problems. A summary can be found in section 4.4.

4.2 Existing Work

The problems presented above are considered in the context of a selection of relevant existing systems. While this is not intended to be exhaustive, it gives a clear picture of the approaches taken in related work. Each of these systems is considered for its approach to problem one, *with regard to references* and problem two on its approach to *coping with copying*.

4.2.1 Java Distribution Technologies

Java RMI has already been discussed, in section 3 in the context of PJRMI. Related distribution technologies are provided by Sun Microsystems for Java that do already use some form of persistence. While the members of the PJama project obviously do not believe that these technologies provide a sufficiently integrated solution for persistence, in comparison with PJama's Orthogonal Persistence for Java, they are considered below.

4.2.1.1 Remote Object Activation

Remote Object Activation (ROA) is supplied with Java RMI in JDK1.2 and documented in chapter seven of the corresponding Java RMI specification [RMI98]. According to this chapter, the aim of ROA is to support long-lived, persistent objects and persistence of client-held references to them, to support communication between them in the face of system crashes. It addresses some of the same issues as PJRMI. Activatable objects are remotely-invokable objects which can be activated on first use; this is similar to PJRMI's support for re-exportation of persistent, remotely-invokable objects and to CORBA Object Activation. An activation description, registered with an `ActivationSystem`, includes information on

the class of the object to be activated and, optionally, a `MarshalledObject` of serialised data. The `MarshalledObject` is used, when an instance of the specified class is created at the point of activation, to initialise its fields. The `MarshalledObject` data is only intended for bootstrapping the activated object; it is serialised for efficient communication, rather than for maintaining an object's changing state persistently. Unlike PJama, there is no support for tracking updates to objects and propagating those updates automatically to the `MarshalledObject`, or to stable storage.

With Regard to References

A client can obtain an activation identity corresponding to a registered activation description. These identities remain valid across multiple program executions so they can be made persistent in some way and then used in subsequent client VM executions. While PJRMI is intended to keep remotely-invokable objects persistent as long as there are persistent references to them, the `Activator`, in comparison, does not track which clients hold activation identifiers. It is intended to run continuously and to maintain the activation descriptors persistently, as long as those descriptors are registered with it. Programmers explicitly unregister activation descriptors when they no longer require them.

Coping with Copying

Since Java RMI is used for communication with activatable objects, the potential for copying large object graphs does exist and, as previously indicated, there is no extra support for dealing with this problem in a manner that is flexible in the long term.

4.2.1.2 Enterprise Java Beans

Enterprise Java Beans [EJB99b] is a component architecture targeted for the development and deployment of component-based distributed business applications.

An EJB server process hosts one or more containers. Each container contains an `EJBHome`, that acts as a factory for creating Enterprise JavaBeans (EJBs), plus one or more EJBs themselves. These EJBs may be session beans or entity beans. A session bean executes on behalf of a single client and is intended to be relatively short-lived. Thus, it cannot persist beyond the lifetime of its EJB container. Although it doesn't represent shared persistent data, it can update persistent data. An entity bean is persistent. It provides an object view of entities in persistent storage, such as an object in a database, and can itself have the lifetime of the corresponding persistent data. Thus, it may persistent across multiple server JVM executions.

With Regard to References

A client initially obtains a reference to an `EJBHome`, via a lookup using the standard Java

Naming and Directory Interface (JNDI). It can then use the `EJBHome` to obtain RMI references (stubs) to EJBs within the `EJBHome`'s container. Clients use Java RMI for communication with EJBs. A client always interacts with an EJB via an interface. Rather than an EJB implementing this interface directly, there is another object, known as an `EJBObject`, which provides a level of indirection. The `EJBObject` is system-generated and implements the interface provided to the client for interaction with the EJB. This allows extra functionality such as transactional support to be provided at the level of the `EJBObject`. Thus, in the simple case, an `EJBObject` just forwards a client's method call on to the EJB while, where transaction support is included, the `EJBObject` wraps the method calls appropriately.

A client can explicitly synchronise the state of an entity EJB with its persistent data by invoking the `ejbLoad` and `ejbStore` methods to read and write data from persistent storage respectively. This is comparable to the load and store methods supported for DCOM components (see section 4.2.8 for more details on DCOM). A container may also invoke these methods; to update the persistent state of an entity EJB when the transaction in which the update took place is committed, for example.

A client-held reference to a session bean is only valid for the lifetime of the container of that bean. If the process hosting that container crashes, the client must obtain a new reference to a new, equivalent session bean after restart. A client-held reference to a remote entity bean is (ideally) valid for the entity's lifetime, which may span multiple EJB server process executions. The reference becomes invalid if the entity is removed or if it is moved to a different EJB container or server. A client-held reference to an `EJBHome` can be serialised and then made persistent; it can later be deserialised and used again as a reference to a remote `EJBHome`.

A client can also obtain the handle of an entity EJB, containing its identity and serialise it to make it persistent. This serialisation can later be translated back into a handle, which can then be used to obtain a reference to a remote `EJBObject` once more. This is obviously intended to be an implementation of the support for persistence of a CORBA Object Reference as a string (see section 4.2.6 on CORBA for more details).

The lifetime of a handle or of a reference to an `EJBHome` actually depends on its implementation, which depends on the persistence mechanism used by the entity EJB's container. It must at least be usable across server restarts. However, the intention is that "Containers that store long-lived entities will typically provide handle implementations that allow clients to store a handle for a long time (possibly many years)." Thus, the problem of long-term dependencies between client and server persistent stores is highly relevant to Enterprise JavaBeans.

A client can explicitly create and remove EJBs from an `EJBHome`. The client uses a method

of the `EJBHome` interface to remove an EJB. If the client tries to use its reference to the EJB subsequently, it will get an `java.rmi.NoSuchObjectException`.

The implications of the EJB specification for dependencies between distributed, persistent stores, is that the onus is on a server to provide remote access to EJBs for as long as required by its clients. The use of Java RMI for communication between client and server implies that the EJB must remain remotely-reachable for at least as long as it is remotely-referenced, since the implicit use of leases in the Java RMI DGC implementation will result in leases being renewed on access to an EJB from a client-held reference for as long as the client holds the reference or client is active. Since there is support for a client to make references and handles for EJBs persistent and to make `EJBHome` references persistent, the implication is that the server EJB should persist as long as there may be a client holding a persistent reference or handle that it can use to obtain access to the EJB, even if the client is not currently active. The support for explicit calls by clients to remove EJBs implies that it is the client's responsibility to decide when an EJB is no longer required. All this leaves little scope for server store autonomy.

Coping with Copying

An entity EJB is a component: it represents an independent business object. The entity object may itself hold references to a large number of dependent objects. Although an EJB must always be passed by reference (i.e. replaced with a stub) when supplied as a parameter in an RMI call, other objects passed in RMI calls between an EJB and its clients may be passed by reference or may be passed by copy.

A "feature" of Enterprise JavaBeans is that all communication between an EJB and its client is made using RMI calls, even when both are instantiated within the same JVM. Thus, all parameters passed by copy in these RMI calls must be serialised and deserialised, even when passed within the one JVM. In fact, local objects *must* be passed by copy in RMI calls between EJBs that are instantiated in the same JVM, to avoid sharing object state between two EJBs, which breaks the EJB's semantics.

The copying of RMI parameter objects raises the issues described in chapter 3 on PJRMI. Because these objects may have large transitive closures of objects and may be passed as a deep copy, they can take a long time and a lot of space to serialise and deserialise.

The Importance of Being Persistent

It is notable that, according to the WWW page introducing the new features of the latest EJB specification [EJB99a], under the heading of "Persistence", mandatory support for entity beans has been introduced earlier than planned "Due to strong demand from the marketplace". Given the popularity of support for persistence in a distributed system, it is clear that there is a need for well-integrated persistence and distribution support that does consider

the implications for long-term maintenance of valuable persistent data.

4.2.2 DPS-algol

Turning to orthogonally persistent technologies, early attempts to integrate orthogonal persistence and distribution included Distributed, Persistent Algol (DPS-algol). Distributed, persistent Algol [Wai88, Wai89] aims to simplify the programming model for distributed applications. It integrates a model of distributed programming with the PS-algol orthogonally-persistent programming model. It maintains location transparency over the distributed data as much as possible. The same syntax can be used to manipulate both local and remote data. Light-weight processes can be started remotely and data objects in a remote location can be referenced.

It is acknowledged that the application programmer may wish to manage the location of data explicitly, for management of resilience and resource utilisation. Thus, a `locality` type is introduced into the programming language for this purpose. A `node` is a locality type that refers to a specific remote machine, while a `loca` provides a way for a programmer to refer to a collection of remote data objects while abstracting away from the `node` that actually hosts them.

Remote procedure calls (RPCs) are used for communication in DPS-algol. A procedure is invoked remotely on an `entry`, supported by the process corresponding to a given process handle. As with procedure calls in PS-algol, parameters are passed by value in RPCs, in an attempt to support something similar to the "blackboard view" of data in PS-algol stores; this is not the same as pass by copy.

The passing of parameters by value applies to pointers, as well as to scalar types. Thus, when a pointer is passed in an RPC, it is replaced with a universally recognisable remote pointer. The difference between a local and a remote pointer is transparent to the programmer. Once a remote pointer has been received at a remote site, any store operations that perform updates to remote referends trigger implicit RPCs back to the pointer's original site.

4.2.2.1 With Regard to References

Once an object has been exported, the intention is that it remains available for remote use as long as it is remotely-referenced. Once the object becomes referenced from an Export Table in a persistent store, it persists.

The persistent store is described as containing a graph of nodes which are collections of local and remote data. Remote pointers are implemented as objects on the heap and can be made persistent, like any other heap object. Once a universal address has been exported for

use in a remote pointer, the corresponding data is expected to persist for as long as a pointer is held to that data, even if the pointer is in a remote address space.

If no abstract machine is currently running over the store when a remote pointer referencing one of the store's objects is dereferenced, the dereference request is redirected to a "perpetual server" process. This server retrieves the required object from the store itself and performs the required operation on it.

No distributed garbage collection has been implemented for DPS-algol but the need for it is acknowledged. The need for a server to keep track of remote pointers is identified, as is the need for a client to inform a server when a pointer it held to a server object has been garbage-collected.

With use of a "perpetual server" to ensure availability of remotely-referenced objects, the implication is that a server store can never escape its obligations to other stores, as long as remote pointers are held to its objects. Since remote pointers can persist, a server can be obliged to maintain objects indefinitely. Subsequent experience with this technology in the COMANDOS project demonstrated that stores become interdependent and hard to manage as a direct result of such obligations. This has been a major influence in the quest for a balance between store autonomy and a uniform model of orthogonal persistence [Atk96].

4.2.2.2 Coping with Copying

Where copying of objects is required between stores, it is explicit. The `transcopy` and `assign` operations are used for this purpose.

The `transcopy` operation uses type to determine what should be copied. The aim is to avoid unnecessary copying and particularly to avoid the copying of a whole persistent store. Some examples of how type influences copying in this case include: immutable base types such as integer, boolean and string are copied; `loca` and `node` base types are passed by reference; process handles are passed as handles to processes on remote machines; images of pixels are deep copied; the top level of a vector or structure is copied while the rest is presumably referenced. Thus, to obtain a complete copy of data structures with some depth, they must be copied incrementally.

The `assign` operation has the same effect on an object as `transcopy`. It copies the top-level of one object graph and assigns this copy to another object, which may be in a different locality.

Such incremental copying, when applied to large, complex graphs of data structures, is imposed on the basis of type. Thus, it is not adaptive over time to changes in graph size or the context in which it is used. High latency costs are incurred on iterations through the

transitive closure of a large graph over the network.

4.2.3 `rx` for Napier

Napier88 [MCC+99], like PS-algol, is an orthogonally-persistent programming language. The work on a remote execution mechanism for Napier88, described in [DRV91], points out the lack of scalability in a one-world model for distributed, persistent systems. It advocates a federated model where the application programmer is fully aware of the distributed nature of the system.

4.2.3.1 With Regard to References

The remote execution mechanism `rx` designed for use with distributed Napier stores avoids the passing by reference of any data, on the grounds that this creates dependencies between stores that necessitate coordination of global stabilisations. Such global stabilisations are avoided on the grounds that they impose an unrealistic requirement for stores distributed across a wide-area network.

4.2.3.2 Coping with Copying

All parameters for an `rx` call are instead passed by copy. Although it is acknowledged that the design of `rx` allows arbitrary amounts of data and code to be copied between stores, no specialised handling of large amounts of data is advocated.

4.2.4 Persistent, Type-safe RPC for Napier88

Coming from the same stable as PS-algol, Napier88 [MCC+99] supports orthogonal persistence. It, in turn, is the predecessor of the support provided by PJama for orthogonal persistence for Java. The support developed for persistent, type-safe RPC for Napier88 [dS96] is comparable to PJRMI and raises the same issues.

4.2.4.1 With Regard to References

Napier88's support for language reflection, dynamic binding and first-class procedures enables the creation of RPC client and server stubs as procedures at run-time.

A server makes one of its procedures remotely-invokable by obtaining a server-side stub for it; this exports the signature of the procedure and the identity of the server itself to a binding

service and has the effect of making the procedure persistent too. The binding service is a trusted entity in the system.

A client obtains a client-side stub by making a *local* procedure call that generates the stub, based on the given signature of the procedure it requires to use. This supports independent, unordered creation of client and server stubs. The first time a client actually makes a procedure call on its stub, the binding service's `import` procedure is automatically called to bind the client stub to an actual server-side procedure. A capability for an exported procedure is returned, along with the address of the server supporting it, allowing the client to go ahead and make RPCs using its stub.

A server can remove support for a procedure arbitrarily. The client will find out that the procedure is no longer exported when its RPC fails. The client can throw away the stub when it no longer wishes to make RPC calls on the server-side procedure. Doing so has no effect on the exported server procedure, on the grounds that other clients may still use it.

Thus, maintenance of persistent data at the server is independent of client use; this means a client-side persistent store may contain references to server-side procedures that are no longer usable.

4.2.4.2 Coping with Copying

In the first version of Napier88 RPC, parameters to Napier88 RPCs are passed by value, to avoid accumulating references, and thus dependencies, between persistent stores. A deep copy is made of every complex value parameter, resulting in whole transitive closures being transferred in RPCs (though there are restrictions on the types that can be copied). No shared subgraphs of data objects are maintained, even between the parameters in one RPC.

Objects in a Napier88 store can be highly interconnected, because of the language's rich type system and the persistence of objects by reachability. To address the problem of avoiding unnecessary copying of large object graphs between persistent stores, "migration by substitution" was implemented for a subsequent version of Napier88 RPC. Application programmers at source and destination must agree on the substitutable objects in advance. Each substitutable object is registered by name. During copying, each object is looked up by value in the substitution table. If it is substitutable, it is replaced with a surrogate. At the destination, the surrogate is used to identify the local value that is equivalent to the original. Parameter objects must either be copied between sites, or substituted with equivalent objects at the destination site. This avoids the creation of remote references at the cost of doing copying and managing substitution. The cost of migration by substitution lies in the registration and lookup of substitutable objects. For one process interacting with a number of other different processes, this is likely to require maintenance of one substitution table

per remote site.

Persistent spaces were also developed as another alternative for sharing objects between persistent stores. These containers of objects are published by a server and copied in their entirety by a client. In this case, it is the application programmer's responsibility not to place object graphs into a persistent space that are too large for copying.

4.2.5 Thor

Thor is a persistent object store developed for use in a distributed system [LCSA99, LAC+96]. It is similar to PJama in that it supports the persistence of objects through reachability from a root object; thus, when objects are no longer reachable, they are garbage-collected. It aims to support good performance for use of distributed Thor objects, even in a wide-area, large-scale distributed environment.

Thor objects are implemented using Theta: an object-oriented, type-safe programming language developed by the Programming Methodology Group at MIT. However, an application does not have to be written in Theta to use Thor. It can be written in a language such as C or C++. A `veneer` of a few procedures can then be used to interact with the Thor store and to make method calls on persistent objects in the store, indirectly via stubs for each persistent type.

Copies of Thor objects are cached at clients, in order to reduce the load on the server and Thor objects may be replicated across multiple servers for high availability.

4.2.5.1 With Regard to References

A client starts a `session` to interact with a Thor store. It runs a series of transactions to perform operations on Thor objects.

An initially volatile Thor object can be created within a client transaction; it becomes persistent if a reference is established to it from an already-persistent Thor object and the transaction is committed successfully at the server. A Thor object then persists at the server as long as it is reachable from one of the persistent server root objects or from a handle of a current session. When it is no longer reachable, it can be garbage-collected.

Distributed garbage collection is managed using reference lists [ML97]. Whenever a client process receives an object reference, whether from its originating site or from a third party, the client adds the reference to its `outrefs` table and sends an `insert` message to the originating site. The originating site containing the referenced object puts the client process into its `inrefs` table, under the entry for the referenced object. Correspondingly, when the

garbage collector local to the client identifies the object reference as garbage, it removes it from its `outrefs` table and sends an `update` message to the originating site. The originating site can then remove the client from its list of processes that reference the corresponding server object in its `inrefs` table. If no other reference is left to the server object, remotely or locally, the server object is then eligible for garbage collection itself. The paper [ML97] focusses particularly on how to deal with cycles in distributed garbage collection, using back tracing.

A client can obtain references to Thor objects within a session, either by looking up a server root object by name or as the result of a method call on another Thor object. However, these references are not valid across multiple Thor sessions. Thus, client use of references is limited to the lifetime of one session. There is no point in a client trying to make such references persistent. A client can, at most, require a Thor object to exist until the end of the session in which the client obtains a reference to it.

Thor objects are stored at a server in an object repository. Though transparent to the clients, there are multiple object repositories and an object can either reside in one or migrate from one to another.

Store maintenance problems exist where Thor objects in one repository hold references to Thor objects in a different repository.

4.2.5.2 Coping with Copying

Copying of Thor objects to clients, done only for the implementation of caching, is limited to the page size, by the Hybrid Adaptive Caching (HAC) cache management scheme used in the Thor implementation. When an object is accessed by the client, the page containing the object is copied from the Thor store to the client cache. To counteract the problem of pages with bad clustering filling the client cache with unwanted objects, hot objects are kept while unused objects on a page may be discarded subsequently from the cache, to make room to copy more pages to the client. Thus, the copying of a large graph of Thor objects, which could be required to support an application's iteration through all the objects of the graph, is done by incrementally copying over the relevant pages.

The problem with copying pages at a time is, as acknowledged in [LAC+96], that the cost of sending the potentially unwanted objects contained in the rest of the page will be significant when used in a WAN or wireless network.

4.2.6 CORBA

The Common Object Request Broker Architecture (CORBA) [OMG99a, OH98] is comprised of a collection of designed-by-committee specifications for middleware. Produced by the Object Management Group (OMG) [OMG], a consortium of over 800 companies, CORBA has a dominating influence on current distributed systems development. It's use of persistence for reliability makes it relevant in this chapter.

Part of the power of the CORBA specifications is that they describe the interfaces for a large range of distributed system services, while leaving a clear separation between these interfaces and their implementation. One of the benefits of this is that, while the interfaces are defined using CORBA's Interface Definition Language (IDL), their implementations can be written in any language with a CORBA binding, including C, C^{++}, Smalltalk or Java, for example. This enables interoperability across a distributed system of applications written in these different languages and incorporation of existing, legacy systems.

The discussion below, on features of CORBA, that are relevant to the problems of this dissertation, is based on the latest formal CORBA specification [OMG99c], unless otherwise stated.

4.2.6.1 With Regard to References

An Object Request Broker (ORB) acts as an object bus. An object implementation accesses services provided by the ORB through an object adapter. Services of the ORB-supported object adapter can include generation and interpretation of object references, method invocation, object and implementation activation and deactivation, mapping of object references to implementations and registration of implementations. An object implementation providing an application service must be associated with a Portable Object Adaptor (POA) to specify what policies are applied to it, and registered with the ORB, before it can be used remotely.

Clients usually obtain object references as parameters or return values from invocations between the client and other objects, or from the OMG Naming and Trading Services.

Persistent CORBA Objects

By default, an object created in a POA is transient: it cannot outlive the POA in which it is created; after the POA has been deactivated, use of an object reference generated from it will result in an `OBJECT_NOT_EXIST` exception. However, if, when the POA is created, it is passed a `LifespanPolicy` set to `PERSISTENT`, the objects created in that POA can outlive the process in which they are created.

The recently adopted Persistent State Service (PSS) specification [OMG99b] is intended to

supercede the Persistent Object Service of the formal CORBAservices specification [OMG98]. It supports the persistence of CORBA object implementations.

Persistence is supported by datastores that may be implemented as, for example, flat files, an object database management system or a relational database management system. A datastore contains a set of storage homes. A storage home contains storage objects. Each storage object contains an identity and a type that defines the state members and operations for its instances.

Storage types, storage homes and catalogs can either be defined using the Persistent State Definition Language (PSDL), which is a superset of the OMG's IDL, or can be defined directly using a programming language; the latter is known as "Transparent Persistence".

An application process interacts with a datastore in a session. An ORB contains a connector registry, which can be used to obtain a connector to a named datastore. A session is established between the application and the datastore, using the connector. The application uses the session as a catalog to look up storage home instances within a datastore, which gives it access to the storage object instances in that storage home. Storage objects can be made remotely accessible by binding their identity to that of a CORBA object.

Persistent References

The representation of the object reference that is handed to a client is only valid for the lifetime of that client. It cannot be made persistent as-is, because different ORBs generate and handle different representations of object references and these references are opaque. The default persistence solution is to convert an object reference to a string; in this form it can be made persistent across multiple client runs or communicated to other processes. An ORB can subsequently generate its own representation of an object reference from the string. Thus, clients can hold references to CORBA objects, store the references in string format and then reconstitute them for use again later.

Maintainability

As long as an object implementation persists, it appears to be always available to the CORBA clients that use it. Like PJRMI, support for activation on first use ensures that, as long as the server is running, the object is active when it needs to be used. The POA, with which the object is registered, defines how it is activated.

However, while an ORB can keep track of outstanding connections between client and server, this only applies to active clients; not to client-held object references that have been converted to string format. This means the server does not necessarily know whether persistent references exist to its CORBA object.

Similarly, even though an object reference can persist as a string and be recreated by an

ORB as a valid object reference, it is not possible for a client's ORB to *ensure* that the state of the referenced CORBA object is available. The PSS specification states that the lifetime of the state of a CORBA object is not visible to its clients. Thus, a client cannot tell whether the implementation of the object it uses is persistent. A client can only call the `non-existent` operation on an object reference to try to determine whether the referenced object still exists: the operation can return true or false, or it can raise an exception if, for example, distribution-related errors prevent the operation from working out whether or not the object exists.

An object reference itself exists until it is explicitly freed with a call to its `release` operation. The `release` call on the object reference has no effect on the referenced object implementation.

According to the text of the CORBAservices LifeCycle Service specification [OMG98], storage management through use of, for example, garbage collection and reference counts, is implementation dependent.

Thus, although there may be, at the application level, an implicit requirement for a CORBA object to be made persistent if an object reference generated for it becomes persistent, there is no support for this in CORBA. It follows that any management of persistent references to create, maintain and limit dependencies between distributed stores is entirely specific to the CORBA implementation being used.

4.2.6.2 Coping with Copying

Whether to Copy

Until recently, CORBA only supported the passing of parameters to IDL-specified methods by value if they were scalar types; objects were always passed by reference. The CORBA specification now acknowledges the utility of copying objects between processes where, for example, the main purpose of the object is to encapsulate data or where the application requires a copy of an object. During specification development, this was referred to as passing "Objects By Value" [MOM98]; it is now described in the formal specification under the heading of "Value Type Semantics".

The criteria for deciding whether to pass a parameter object by reference or by copy is based on the signature of the operation to which it's being passed. If the parameter type in the operation's signature is a CORBA interface, then the object to be passed as this parameter will be passed by reference. If the parameter type is a CORBA value type then the object will be passed by copy.

An interface in CORBA is comparable to a Java interface. It declares the signatures of a

collection of operations but does not define their implementations. A value type in CORBA is comparable to a Java class, in that it describes a set of operations and some associated state.

Thus, passing by reference or by copy is not defined on the object definition itself, but rather on the parameters in the method signature of operations that use the object. However, in order for the programmer to have the *option* whether to pass an object by reference or by copy for a given operation, the object must have been defined with a CORBA interface.

This means that object passing policy in CORBA is defined statically, but with regard to the application code that uses the object, rather than with regard to the object definition itself. The same policy is not enforced on an individual object, without any consideration of the context in which the object is used. By setting the policy using the operation signature, different applications (or even different operations within the same application) can pass the same object between processes in different ways. If an object implements an interface, it can be passed as a copy of the object implementation in one operation invocation and as a reference to its interface in a different one.

The object to be copied can have complex state, with arbitrary graphs, recursion and cycles. During copying, shared subgraphs are preserved between the parameters involved in one invocation. However, the copy shares no state with its original. At its destination, the object copy has a separate identity from the original object.

What to Copy

The ORB implementation defines the code for marshalling parameter and return values at their source and unmarshalling them at their destination. Thus, the manner in which an object graph will be copied from one process to another is dependent on the marshalling code of a specific ORB implementation.

Value types are allowed to override the standard ORB marshalling with their own code for marshalling and unmarshalling their own state. However, this is regarded as exceptional, rather than the norm, intended only for integration of existing "class libraries" and other legacy systems.

The CORBAservices Life Cycle Service specification describes how an object can be copied in a distributed system; so it could be used in a marshalling implementation. To take advantage of this service, the object implementation to be copied must support the interface `LifeCycleObject`, which supports the operations `copy`, `move` and `remove`. A simple object, with no references to other objects, provides its own implementation for each of these `LifeCycleObject` interface operations. As an appendix to the Life Cycle Service, a Compound Life Cycle Specification is provided, which defines how a compound life cycle operation is applied to a graph of related objects, given a starting node.

The CORBAservices Relationship Service is used by the Compound Life Cycle Service to inform the copying of an object graph, based on the relationship declared between objects of the graph. Consider the example of a folder object that contains a document object. The relationship between folder and document is defined using three objects. The folder is associated with a `ContainsRole` object, while the document is associated with a `ContainedInRole` object. A `containment` relationship object connects the two roles. Thus, three Relationship Service objects represent the relationship between the folder and the document. The Compound Life Cycle specification makes two passes over an object graph, initially to analyse the relationships between objects to determine what objects of the graph should be copied and subsequently to actually perform the copying. An object with a `ContainsRole` should be deep copied, to include the objects that it contains, while an object with a `ContainedInRole` is shallow copied for the purposes of this relationship (though presumably if the latter object has a `ContainsRole` in relation to a different object, it can actually still end up being deep copied ultimately).

4.2.7 GemStone

GemStone is a commercial implementation of persistence. It was originally developed in Smalltalk, now available as GemStone/S, and has now also been developed in Java, as GemStone/J [Gem99]. The benefit of years of experience with Smalltalk are evident in the maturity, sophistication and scalability of the current systems. A Persistent Cache Architecture maintains the illusion of shared memory over server processes (Smalltalk execution engines or JVMs respectively) for high-performance, server-side persistence. Server objects become persistent by reachability from server-side named root objects. Client applications must establish a session with a GemStone/J server in order to get access to its objects and services. They then initially get access to server objects by looking them up by name in the Object Name Service. When a client wishes to make changes to persistent server objects, it must make the changes within a transaction; the changes become persistent if the transaction commits successfully at the server. Distributed clients communicate with the server using one of a range of technologies: for GemStone/S this includes Smalltalk, Java and CORBA, while for GemStone/J this includes Java RMI, Enterprise JavaBeans and CORBA.

The GemStone/J server has been implemented for scalability. Two models provide this scalability in different ways, depending on the requirements of the client.

1. For scalability through use of threads, a server object is instantiated in one server-side JVM and shared by multiple clients using multi-threading.

2. For scalability through use of persistent objects, multiple instantiations are made of

one server object, each in their own server-side JVM, providing unshared access for each client.

The problems of this dissertation are considered below, largely in the context of GemStone/J.

4.2.7.1 With Regard to References

GemStone/J supports CORBA through use of the VisiBroker for Java ORB [Gem98a]. The Visibroker ORB supports communication between CORBA client and server objects. To use a Java object as a CORBA object, the Java object's class must implement a CORBA-supported interface. CORBA objects can be activated using Visibroker's Object Activation Daemon. A client can then obtain references to the CORBA objects supported by the GemStone/J server.

GemStone/J also supports Enterprise Java Beans (EJB). JavaBeans are created at the GemStone/J server. A client communicates with a JavaBean either via a Remote Adaptor supporting the JavaBean's interface, or through use of JavaBean events. However, a Remote Adaptor appears to be valid only for the lifetime of the current client-server session[1]. Thus, even if a client has its own persistence support, it is of no benefit to the client to make the Remote Adaptor persistent beyond the lifetime of the session in which it was obtained from the GemStone/J server.

The garbage collection criteria for when a server object is no longer reachable are not known, since details of GemStone's GcGem garbage collector have not been made public. Thus, although there is apparently some tracking done of the objects accessed by clients, it is not clear whether the server is obliged to maintain remote access to GemStone/J server objects as long as they are in remote use. However, since GemStone/J only supports persistence at the server side, the implications of persistent references and the complications of having a mixture of persistent and non-persistent clients, as encountered in PJRMI, are not addressed.

4.2.7.2 Coping with Copying

The potential exists for the copying of large object graphs between a GemStone server and its clients. GemStone/J supports classes for large collections, which it describes as scalable containers because of the attention paid to their scalability in the GemStone/J implementation.

VisiBroker for Java 4.0 conforms to CORBA 2.3. Thus, it includes support for passing

[1]This is certainly the case with their equivalent in GemStone/S, which are referred to as `forwarders`. A message sent to a `forwarder` after the end of a session results in a "defunct forwarder" error [Gem96].

objects by value. Its use for communication between clients and servers in GemStone/J has the implications described in section 4.2.6 on CORBA.

Where EJB is used, parameters to remote message invocations between a client and a server JavaBean may be passed by reference or by copy.

- Java scalar type values and Strings are passed by copy.
- A remote adaptor is marshalled in place of an object that implements the interface `GsRemoteIF`.
- An application can choose at runtime whether to copy an object or replace it with a remote adaptor, if it implements the interface `GsExtendedRemoteIF`.
- If the object does not implement a remotely-enabled interface, but it does implement the Java interface `java.io.Serializable`, then the object is passed as a deep copy of its object graph.

The documentation for Gemstone/J Distributed JavaBeans [Gem98b] explicitly warns that "Where the entire object graph must be returned as a copy, performance is likely to be of concern in the case of large collections or large object graphs." It is recommended that large object graphs are accessed by remote reference rather than by copying, for the reason of maintaining sharing as well as communication performance, but ultimately it is left to the application programmer to try to avoid passing large object graphs between client and server.

Of the support described above, the interface `GsExtendedRemoteIF` is of most interest to the author. An object that implements this interface must define its only method `asCopy()`. This method is called during serialisation and returns a boolean indicating whether the object should be copied or not. It could be implemented, for example, to pass by copy normally but pass as a remote adaptor if its size is larger than some limit. This, unlike other existing systems, does provide support for a run-time, and therefore adaptive, decision to be made about whether or not an object should be copied between sites. The programmer must, of course, have defined the object from the outset to implement the interface `GsExtendedRemoteIF` in order to have that run-time choice.

It is also interesting to note that while GemStone/J does not currently have any other dynamic way of controlling object copying, such support was implemented for management of copying for replication in GemStone/S. It has recently been brought to the attention of the author that this support exists in GemStone/S and is comparable to one of the solutions presented later in this dissertation, in chapter 8. In GemStone/S, copying for replication

is controlled by specifying the level (depth) to which objects in a graph should be replicated, after which object stubs are created to represent the non-replicated lower-levels of the graph [Gem96]. Subsequent access to the stub objects results in them being copied on demand.

4.2.8 DCOM

Because CORBA and DCOM currently have much influence in commercial distributed systems development, both are considered in this chapter. DCOM has been developed by Microsoft, using its COM component model, to support use of components across distributed processes [RE98, Ses98]. A component is a module of software, designed to do a specific task, and with a well-defined interface. The aim is to be able to compose a system from components to provide, for example, support for electronic commerce for banks, travel agents, credit card services, etc. (EJB, as presented in section 4.2.1.2, is comparable to DCOM, in that it provides a component model for Java.)

4.2.8.1 With Regard to References

COM components can be made persistent in flat files, architected files (e.g. sequential or indexed files) or relational databases. Server-side support uses the `IMoniker` interface, implemented for a specific persistence mechanism, for maintaining an association between a name, which can be considered a persistent identifier, and the corresponding persistent state of a component. The interface includes a `BindToObject` method that, given a name in the appropriate naming convention for the persistence mechanism, returns a reference to the corresponding persistent component. It's implementation instantiates a new component of the correct type and relies on that component's implementation of the `IPersist` interface to populate it with the persistent state.

Since component references are only valid for the lifetime of the process in which they are generated, there is no point in making them persistent. However, the `Moniker` for an object can be made persistent, using the `IMoniker` interface method to convert a `Moniker` to a string. Another method of the same interface can be used to convert the string back to a `Moniker`, after which its `BindToObject` method can be called to establish a reference to a component containing the corresponding persistent state once more.

Client applications can obtain remote references to components as proxies, generated from Microsoft's Interface Definition Language. Communication between distributed components is done using Microsoft's remote procedure call support (MS RPC). Management of referenced, remote components is done explicitly in application programs. When a ref-

erence is established, an explicit call can be made, by the application programmer, to the component to inform it that it is being remotely used. When a reference is no longer needed, an explicit call can be made to the component to inform it that one less reference will use it from now on. Additionally, a pinging protocol is used by client's to regularly inform a DCOM server that it is still alive, in order to keep alive its connections to the server's DCOM components. However, this is only applicable for the lifetime of the client.

4.2.8.2 Coping with Copying

Marshalling of parameters in calls between components is usually done in the code of the IDL-specified client proxies. While the potential for copying large amounts of data between distributed components does exist, no additional support is provided for handling large data volumes in any specialised way.

4.2.9 Arjuna

Arjuna aims to provide support for building fault-tolerant, distributed applications, using persistence for reliability and transaction recovery. Several products are available from Arjuna Systems. They benefit from years of experience doing research on support for fault-tolerant, distributed applications in the Arjuna project at the University of Newcastle in the UK. This section focusses mainly on Arjuna Integrated Transactions (AIT) for Java [Arj99], on the grounds that it is representative of the approach of Arjuna solutions.

AIT provides support for use of objects in transactional applications, with persistence to aid reliability and recovery. The state of an object is marshalled and stored in a file or database for persistence. Clients obtain references to these objects in the form of stubs. AIT objects may have one of three flavours. If they are recoverable and persistent, their state is tracked for recovery and maintained on stable storage for use over multiple program executions. If they are only recoverable, then they cannot have a lifetime beyond the current program execution, but their state is tracked within transactions for recovery purposes. If they are not recoverable or persistent, they do not survive program crashes or shutdowns.

4.2.9.1 With Regard to References

Like GemStone, as described in section 4.2.7, AIT supports two models of server object usage. Multiple clients may use one shared, persistent object at the server. Alternatively, multiple clients may each have their own replicated copy of the persistent object at the server. The first of these two models is the default.

Remote use of Java objects in this context has the same reference management issues as for standard Java RMI, except for the following extra transaction-related support. A reference to a persistent object may be created in a transaction or, if stored as a CORBA Inter Orb Reference (see section 4.2.6) it may be re-established from a string form of the object reference. A call may be made explicitly to destroy a server object or a referenced object may become unreachable, in the course of a transaction. However, it will not be garbage-collected until the transaction commits successfully, in case it is necessary to reestablish a reference to the object in the course of an abort of the transaction instead. For long-running transactions, the server is thus obliged to maintain the objects used for the lifetime of the transaction.

4.2.9.2 Coping with Copying

Remote use of Java objects has the same issues in this context for copying of object graphs between distributed sites as for standard Java RMI.

4.2.9.3 The Arjuna Project

Much work has been done in the context of the Arjuna project on support for fault-tolerant distributed systems, with particular focus on replication of persistent objects, to provide reliability and high availability in the face of the inevitable distribution-related failures. This has included work done on integration of replication support with transactions [LS99b] and with caching [LS99a]. Arjuna's focus does not encompass consideration of the long-term implications of dependencies between persistent stores. The onus is left on the server to provide remote access for its clients as long as it is needed. While large object graph copying is obviously an issue with replication and caching technologies, only the management of interdependencies has been considered for replication of large object graphs [LS96], while the cost of the actual copying does not appear to be addressed.

4.2.10 PerDiS

The aim of the Persistent Distributed Store (PerDiS) project is to provide support for distributed, collaborative engineering applications [FSB+98]. The significant feature of such applications is that they share large volumes of fine-grain, complex objects across wide-area networks. PerDiS aims to provide integrated, automated support for this.

4.2.10.1 With Regard to References

PerDiS attributes distributed CAD application problems, of abysmal performance and lack of server scalability, to client use of remote references to access server objects. To deal with these problems, it instead provides the illusion of distributed shared memory (DSM) across the network, with support for consistency and concurrency control.

Objects persist by reachability from named root objects. Multiple persistent stores cooperate to provide the persistence of the objects in DSM. Currently, this seems to be implemented as one cluster per file on disk. Clusters in one store may hold references to clusters in another store. However, it is not possible for individual stores, that can in theory contain one or more, possibly replicated, clusters of persistent objects, to be managed separately and thus autonomously.

Local caching of remote objects is implemented using either

- explicit calls, made by an application navigating through an object graph, to identify the objects in that graph to be cached locally, or
- automatic faulting of pages of the cluster's storage on access.

All updates are done in the context of a transaction and applied to the, possibly persistent, cache and to disk when the transaction commits.

The problem of dangling pointers, caused by deleting an object that is still reachable, is identified. The PerDiS solution is automatic storage management, using the Larchant distributed garbage collection algorithm [FS98].

Every application process interacts with PerDiS through the interface provided by a User Level Library (ULL). The ULL interacts with the single PerDiS daemon running on its local machine. The PerDiS daemons cooperate to support DSM. The ULL is responsible for detecting new inter-cluster pointers when they are established by the application. A stub is created, to be associated with the pointer, and a message is sent to the referenced cluster, where a corresponding scion is created, to be associated with the referenced object in that cluster.

The PerDiS daemon, running on each machine hosting application processes, does garbage reclamation by marking all the objects in locally-cached clusters that are reachable from persistent roots or scions. Non-marked objects can then be deleted. The implication is that objects must exist as long as any reference to it exists, whether the reference is local or remote (the latter represented by the existence of scions).

4.2.10.2 Coping with Copying

Given the two methods described above, for identifying which remote objects or remote pages of objects should be cached locally, there is support for controlling the amount of data copied between distributed sites. The application programmer may control the caching, and thus the copying, of the object graphs they expect to use. They must be aware that calling `hold` on very large graphs of objects will result in long and expensive copying operations, in order to bring them into the local cache. Alternatively, when pages are automatically faulted, their copying is batched to the size of a page on disk. The efficiency of this mechanism depends on how well the required data is clustered on these pages.

4.2.11 FlexiNet

A product of the ANSA collaborative research programme on distributed systems, FlexiNet is intended to demonstrate the ANSA architectural principles at work [HAN99b]. The flexibility of this system is in its ability to support a range of RPC mechanisms. These mechanisms take the form of binders, that support different combinations of layers of the communication protocol stack at client and server. This enables a plug-and-play philosophy, covering aspects of the protocol stack including naming, serialisation and transport protocol. A Trader, object location service, mobile object workbench and persistent information space are just a selection of FlexiNet's other services.

4.2.11.1 With Regard to References

FlexiNet rejects passing objects purely by reference in remote method calls, because of the performance overheads of following such references over the network when access to the referenced object is required. FlexiNet also rejects the Java RMI model of passing objects by value normally and by reference if they extend the interface `java.rmi.Remote`. The rejection in this case, in agreement with the author of this dissertation, is on the grounds that it is not reasonable to pass an object by value in some cases and by reference in others. Instead, FlexiNet takes the same approach as CORBA's Value Type Semantics 4.2.6. The decision on whether to pass a parameter object by value or by reference is based on the declared type for that parameter, in the definition of the operation to which the object is being passed. Thus, if the object is passed as an interface, then it will be passed by reference; otherwise it will be passed as an object and thus by value.

4.2.11.2 Coping with Copying

FlexiNet has the potential for large object graph copying. It has support for persistence and for management of clusters of objects. However, no extra support is provided for handling communication of large object graphs between sites.

4.3 Related Work on Alterative Approaches

There are many other systems that support both distribution and persistence, in some form. However, the descriptions of those above demonstrate the degree of general awareness in existing work of the issues explored in this dissertation and the kind of steps, if any, that have been taken to deal with them. Brief descriptions of some other related work are now presented, to demonstrate alternative approaches, including object substitution, object movement and network objects.

4.3.1 Approaches of Database Systems

Database systems are increasingly being used over wide-area networks, laying them open to the issues of this dissertation. Oracle have introduced support for an Internet database, Oracle 8i [Ora99], that has support for SQLJ and for use with JavaBeans and CORBA. The Jasmine Object Database [KDM99] is a good example of the approaches being taken for support of distributed object database access. Its `WebLink` component supports inclusion of ODQL statements in web pages, with the results of queries being presented as "exploded" web pages displaying object values. There is also support for Java RMI communication from Java client applications to an application server using the `pJ` Java persistence layer for Jasmine. None of these systems address the potential problems of communicating large object graphs from server to client and, since most of them use Java, they suffer from the same lack of flexibility with respect to remote object access that has been described above.

4.3.2 Object Substitution

Work on Octopus [FD93] and object migration by substitution [dSA96] has tried to address the problem of copying large object graphs by limiting the copy to those objects in the graph that have no equivalent at the copy's destination. The Octopus mechanism supports the cutting of bindings within the closure of an object graph to be copied, and the rewiring of the partially-copied object in another context.

Although lacking the elegance of the dynamic linguistic reflection mechanism of Octopus

and Napier88, the use in Java RMI of `readObject` and `writeObject` methods associated with copied object classes, can provide a similar ad-hoc solution. Java Object Serialisation allows a programmer to override the default serialisation implementation with specialised marshalling, defined on a per class basis. A programmer can choose to omit or replace certain fields of an object during marshalling. These fields can then either be left with a default value during unmarshalling, or set to reference local resources at their destination.

4.3.3 Object Movement

Emerald supports location-transparent use of Emerald objects distributed across a local area network [BHJ+87]. It considers the problems of maintaining references and moving objects in this context, with the aim of supporting efficient inter-object communication.

Mutable Emerald objects are passed by reference in remote invocations, to preserve consistency. However, Emerald tries to avoid remote references to invocation parameters by, where possible, moving the parameters to the site of the callee.

Immutable objects are moved automatically and a programmer can explicitly request movement of an object, based on their knowledge of the application, using the "call-by-move" parameter passing mode. It is acknowledged that the moving of objects between sites does depend on their size and usage. Moving the object whenever it is passed as a parameter to a remote invocation will be of benefit if it is used multiple times by the destination site but will become inefficient if it keeps being moved between multiple sites that are using it. The implication seems to be that only small object graphs should be moved, while larger ones should only be passed by reference.

4.3.4 Obliq and Network Objects

Obliq is a language developed for distributed, object-oriented computation [Car94]. It is implemented using Modula-3's Network Objects [BNOW93]. Oblique makes a point of avoiding *automatic* copying of object state between sites. Network references to objects are usually passed instead. Values can be transmitted by copying if required; any references held to other objects are replaced with network references during transmission. The Network Objects system provides a general purpose mechanism called `Pickles` for marshalling object graphs of arbitrary complexity.

4.3.5 Other References to Related Work

Further references to related work are made in comparison with the solutions of this dissertation: with regard to references in section 6.3.4. Future work in chapter 11 includes references to related work that could influence and benefit further development based on the solutions in this dissertation.

4.4 Summary

A summary of the approaches taken by existing systems to this dissertation's issues is presented below, firstly *with regard to references* and secondly on their approach to *coping with copying*.

4.4.1 With Regard to References

The lifetime of references to remote objects falls into two main categories: those that are only valid within one client program execution and those which may be used across multiple program executions. Objects that represent a reference to a remote object (i.e. stubs) are usually only valid within one client program execution. Systems including Thor, CORBA, GemStone and DCOM take this approach. However, a large number of systems do support persistence across multiple client program executions of a reference, but only in the form of a string identifier. CORBA IORs, DCOM Monikers and EJB handles to entity beans are three examples of client-held representations for remote objects that are converted to a string to be made persistent and, in a subsequent client execution, can be translated back from a string and used to try to obtain a reference to the corresponding remote object once more.

The influence of clients on the lifetime of remotely-accessible objects at the server varies widely. At one end of the scale, CORBA's Persistent State Service specification makes it clear that the persistence of a service is not made visible to a client, never mind influenced by it. Some systems track references held to services, to maintain those services while they're used, but only for the lifetime of the current client program execution. Those systems that support conversion of a reference to a string for persistence cannot expect a server to be able to track what references are held by clients once they are strings though, in the expectation that the services will be maintained as long as the string identifiers for them persist. As an alternative, DCOM is an example of a system that allows the programmer to make an explicit call from client to server to ensure a service is maintained for the client. At the other end of the scale, systems including DPS-algol and PerDiS take the "integrated persistence" approach of tracking references to remote objects and obliging a server to maintain

its services as long as they are remotely used.

The integrated persistence approach is the one taken by PJRMI, as described in chapter 3. It attempts to avoid lack of referential integrity between distributed stores by ensuring a remotely-invokable object persists as long as a client holds a reference to it. Its drawback is the consequent lack of autonomy and thus lack of long-term maintainability, because of the dependencies created between stores supporting persistent connections.

4.4.2 Coping with Copying

Support for coping with the copying of object graphs across a distributed system tends to be either inflexible or non-existent.

The criteria for whether or not to copy an object varies. Some systems base the decision on the object's type. Java uses interfaces including `java.io.Serializable` and `java.rmi.Remote` for this purpose. GemStone/J's EJB support is similar but, for added flexibility, it also provides the ability for the programmer to make the decision at runtime when the interface `GsExtendedRemoteIF` is used. CORBA Value Type Semantics base the decision about object copying on the declared parameter type for the operation to which a parameter object is passed. DPS-algol requires an explicit call to be made by the programmer in order to make a copy of a data structure.

Where support for object copying is provided, there is usually limited or inflexible control over the proportion of an object graph that is actually copied. A common approach, taken by Java RMI, EJB and DCOM for example, is to make a deep copy of the full transitive closure of a given object graph. No consideration is given to the handling of large object graphs at all. As an alternative, some systems leave it to the programmer to specify exactly what parts of an object graph should be copied. PerDiS provides a mechanism for the programmer to iterate through their object graphs making an explicit call on each object in them that they wish to be copied. The persistent spaces solution for Napier88 RPC requires the programmer to explicitly place copyable objects into the persistent space. By contrast, the migration by substitution for Napier88 RPC requires the programmers at all sites involved to agree and register the objects that are substitutable: i.e. the parts of the object graph that should not be copied.

There are more implicit, controlled-copying mechanisms though. Incremental shallow copying is enforced by DPS-algol for data types that can hold references to other data objects. PerDiS and Thor both incrementally copy pages between sites. GemStone/S, which copies object graphs for replication purposes, has support for limiting the copy to a specified depth of the graph initially; remaining objects of the graph are subsequently copied on demand.

A solution is needed which doesn't leave the decision on how much to copy entirely to the programmer, since they may not be fully aware of the actual number of objects reachable from the object they wish to use[2]. This solution should be flexible enough to work well with long-lived objects. Given that such objects may be used by different applications and in different contexts over time, it is not desirable to require that support for whether and how much of an object graph to copy should be hard-wired into the object's type definition. More dynamic control is needed, on how much of an object graph to copy, that is adaptable over time to the size of the object graph and to the context in which it is used.

4.5 Influences of Related Work on Solutions

To avoid hardwiring the specification, of whether and how much of an object graph to copy, into an object's type definition, support is needed which promotes a separation of concerns. There are existing technologies that advocate a separation of concerns in the provision of distributed systems support. Some examples of these are described here in sections 4.5.1, 4.5.2, 4.5.3 and 4.5.4. However, while such technologies are more likely to provide the flexibility for handling persistent objects throughout their lifetime, they do not address directly the concerns of this dissertation on how to control the copying of large object graphs between distributed sites. The technologies of section 4.5.5 do consider how to control volume of data communication, with an emphasis on quality of service, but these tend to be at the lower levels of transport protocols. The aim of the solutions in this dissertation is to provide control over object graph copying at the application level.

4.5.1 Spring Subcontracts

The Spring system is a distributed operating system that provides a platform for supporting distributed applications. It promotes separation of concerns by supporting specification of a remote method invocation mechanism in a `subcontract`, separately from the objects to which it applies [HPM93]. A `subcontract` implements an interface of operations that are called at significant points in communication between distributed sites; such as at the point of marshalling and unmarshalling RMI parameters, for example. The application programmer chooses from one of a selection of pre-defined `subcontracts` or defines their own and applies it to a Spring object at the server. The Spring platform makes the appropriate calls to `subcontract` operations at client and server. Applying a `subcontract` to communication with a Spring object is largely hidden from the client. The default marshalling operation

[2]This is particularly likely to be the case when components are being used that have been developed by a third party.

moves a parameter between sites. An alternative marshalling operation provides support for copying where it is required instead. `Subcontracts`, as presented in [HPM93], have been defined for replication, access to clusters and caching.

4.5.2 CORBA

The separation of policy specification from application code is achieved through the association of a policy with a Portable Object Adaptor (POA) in CORBA. A number of policy objects are created and associated with a POA. These policies then apply to all objects registered with that POA, to influence, for example, marshalling of requests and activation of object implementations.

When an object reference is generated by an ORB, the ORB implicitly associates it with one or more policy domains, thus imposing certain policies on use of that object reference. Any conflict between the policies set on an object reference and the policies that apply to the referenced object implementation must be resolved. The specification does not yet include interfaces for management of CORBA policy domains though.

4.5.3 GARF

GARF supports the development of reliable, distributed object-oriented applications by providing a library of abstractions for concurrency, distribution and reliability [GGM96]. It promotes a separation of concerns by encouraging the programmer to write the code for their application task separately from the code concerned with the abstractions supported by the GARF libraries. The latter code is written in the form of behavioural objects, also known as meta data objects, which are either `Encapsulators` that wrap the objects to be used remotely or `Mailers` which support communication between the `Encapsulators`. Support for replication, for example, is provided by an `Encapsulator` while support for ordered message delivery to replicas, for example, is provided by a `Mailer`.

In the assessment of GARF, it is noted that while dynamic establishment of the association between application objects and behavioural objects is currently supported, it is not necessary. Static, "once for all", association is considered sufficient, except for open applications such as operating systems, which are outside the scope of GARF. (The author of this dissertation would argue that use of PJama in a distributed system is categorised as an open application in this case.)

4.5.4 A Framework for Policy Bindings

Øyvind Hanssen has been developing a framework for setting quality of service (QoS) policies on bindings created for communication between distributed sites [HE99]. This work is based on the FlexiNet architecture developed as part of the ANSA Architecture for Open Distributed Systems [HAN99b], as described in section 4.2.11. The aim of the framework is to provide a clean separation between, on the one hand, definition and dynamic setting of the QoS policy to be used and, on the other, the code of the distributed application to which the policy applies.

A policy in this case refers to a combination of properties associated with the communication mechanisms at client and server, which may include transport protocol, transparency management and resource management. A policy is negotiated between client and server and then applies to all communication between them for as long as is defined in the policy; probably the duration of a transaction or session. Like the work described above, the separation of policy definition from application code is supported. Like the work in mobile computing (see below), the intention is that the dynamic choice of QoS policy should allow communication between sites to be adaptive to the current distributed environment. A policy for logging has been implemented for this framework so far [Han99a].

4.5.5 Mobile Computing

The issues raised in this paper are of relevance in the domain of mobile computing. The need for a distributed application to be adaptable to the current execution context is of particular importance in this highly variable domain. Mobjects [WB95, WB97] focusses on the need for distributed applications to be able to find out information about the environment in which they are running, with a view to adapting communication policies between mobile host and server in an effort to meet quality of service requirements. Changes in, for example, the network connectivity of a mobile host and in the range of services (e.g. printing) currently available to it are intimated to an application as `EventObjects`. Odyssey [NPS95, NSN+97] has a similar model for allowing an application to register interest in notifications about changes to a specified resource, including the acceptable bounds in which the resource can be used and an upcall procedure to be called to adapt behaviour. The impossibility is acknowledged of a system providing support for mobile data access that is appropriate for every application running in every environment: thus, service guarantees are not provided. What is provided is application data filtering that is adaptive to the current network connection and application requirements. In [NPS95], a comparison is made between a video playback application and a video scene editor; they both work over the same data but, while the player can afford to drop frames when bandwidth is low, the editor needs to be able to

display every frame to the user to ensure accurate editing. The more recent work on Odyssey requires `wardens` to be written for every type to manage fidelity of data between client and server. Doing data filtering in order to limit the amount of network bandwidth or destination resources used is comparable to using policies for limiting the number of objects transferred across the network from a persistent store but the ability to filter tends to be very type or protocol-specific.

Chapter 5

Research Issues to be Addressed

This chapter summarises the research issues of this dissertation, to set the scene for the presentation of solutions.

Orthogonal persistence is intended to ease the programmer's job by providing support for data management integrated with a programming language. The simplicity of the orthogonal persistence model argues for its use in distributed systems, in order to make life simpler for the application programmer.

Support can be developed for interactions between persistent objects in distributed stores. Persistent objects in one store can hold references to persistent objects in another store. Persistent objects can also be copied from one store to another. However, such support reveals problems with combining orthogonal persistence and distribution.

As described in chapter 3, PJRMI supports persistent, remotely-invokable objects and persistent references to them. It attempts to maintain the illusion of persistent connections between stores for as long as they are required. However, PJRMI demonstrates the two important problems which are the focus of this dissertation.

5.1 Problem One: With Regard to References

The first problem is in the provision of this illusion of a persistent connection between stores. Distribution-related errors easily break the illusion. In an open system, it can be difficult to determine when an object should become persistent by remote reachability. In the long term, persistent references to remote objects threaten the maintainability of the persistent stores involved.

With regards to this problem, existing related work typically avoids the issue completely.

It may force the application programmer to ensure that client programs explicitly establish references to remotely-invokable objects every time they are run. It may allow the programmer to make references to remote objects persistent in the form of string identifiers; but with no requirement that services be maintained as long as references for them in the form of string identifiers persist. Where existing work does address the first problem, it obliges servers to maintain their services for as long as they are remotely used. The problem with this approach is that the server can suffer from having to maintain resources indefinitely, if it cannot determine that a client no longer needs them.

Chapter 6 presents solutions for a workable compromise. It explores the issues associated with extending persistence by reachability across a distributed system. Persistent references to remote objects are still supported, but the intention is that they can only be preserved for use within one lifetime of an application. Application leases, set on remote use of persistent objects for the duration of a distributed application's lifetime[1], limit the use of remote references. They provide a solution that compromises on reliability of references in favour of greater store autonomy.

5.2 Problem Two: Coping with Copying

The second problem is raised by copying object graphs between stores, as happens, for example, when an object is passed by copy as a parameter in an RMI call. Large object graphs tend to build up in persistent stores over time. In a long-lived system, assumptions are more likely to change about the size of an object graph and the context in which it is used, during its lifetime.

Some existing related work ignores this problem, by assuming that the programmer is aware of the size of object graphs that they copy between sites and is happy to cope with the costs of copying large object graphs when this does occur. Other work forces the programmer to explicitly indicate which objects of a graph should be copied and/or which should not, on a per object basis. Alternatively, the programmer may have no control; copying may be done between sites only in the implementation at the level of pages rather than objects.

Existing work does demonstrate that it is possible to separate policy for object usage from object definition. This sort of flexibility needs to be applied to the handling of object graph copying. The GemStone/S support for limiting the depth of an object graph copy demonstrates that it is possible to adapt to changing object graph size over time. This sort of adaptability is needed for controlling object graph copying, with greater choice for how that control should be achieved.

[1] A distributed application's lifetime may span multiple store shutdowns and restarts.

Object-copying policies provide the solution. Chapter 7 presents the motivations and assumptions behind the use of object-copying policies for persistent applications. The design is described in chapter 8, with greater detail included at the implementation level in chapter 9. The evaluation in chapter 10 concludes that use of these policies does ensure adaptability, over time, for the copying of object graphs between persistent stores to deal, in particular, with the problem of how to handle large graphs of persistent objects in a distributed system.

Chapter 6

Persistence by Reachability across a Distributed System

6.1 Introduction

The simplicity of the orthogonal persistence model argues for its use in distributed systems. By removing the burden of explicit data storage management, orthogonal persistence support leaves the application programmer free to focus on the details of the application task and the challenges of distributed application management, rather than having to juggle the concerns of all three simultaneously. In theory, applying principles of orthogonal persistence to a distributed system means that, to ensure that persistence remains orthogonal to type, it should be possible for an object of any type to become persistent; even if the object is actually of a proxy type that holds a reference to an object in a remote process. It also means that, where the determination of an object's persistence is by reachability from root objects, there is a requirement to ensure referential integrity: once an object becomes persistence reachable, even from a remote VM, that object and all the objects it references, directly and indirectly, will persist.

Within one process running over a persistent store, it is possible to guarantee the consistent, stable state of persistent objects. However, such attempts to extend orthogonal persistence, from a single process to the less reliable and less controllable world of a distributed system, sacrifice consistency guarantees (and the integrity of object references, in particular) in the persistent stores involved.

The *illusion* of a persistent connection can be provided, as demonstrated by the support for persistent remote method invocation for Java (PJRMI) described in chapter 3. However, there are several problems with maintaining this illusion.

1. It is unrealistic to assume that, just because a reference to a remote object has been made persistent, it will always be possible to access the remote object successfully. Distribution-related errors caused by process crashes and network delays or failures are unavoidable and easily break the illusion of a persistent connection.

2. It can be difficult to ensure that the remotely-referenced object exists for as long as it is required. Extending persistence by reachability across a distributed system implies that if an object becomes persistent and it holds a reference to a remote object then the remote object must become persistent too. It can be difficult to determine when, where and how an object should become persistent by remote reachability though.

3. A long-term problem exists with persistent connections between distributed objects: they threaten the maintainability of the persistent stores involved. A store does not have the control to maintain a consistent state over its objects and to garbage-collect those that it no longer wishes to support, if it is obliged to provide remote access to objects for as long as references are held to them from other stores. By the same token, a store does not have control over the integrity of its references when it holds a reference to an object in a remote store, making it dependent on the remote store for its own referential integrity.

A range of solutions have been considered for these problems. The emphasis on the solutions is that they be realistic, rather than idealistic. The appropriateness of a solution for a distributed, persistent system depends on the priorities of the application programmer(s) that develop and maintain the system.

Thus, an application programmer must choose which of two issues is more important for their persistent, distributed application: a reliable, consistent, local persistent store or a simple model of programming with automated storage of objects, even when those objects are proxies for objects in a remote store. Realistically, because of the intrinsic lack of reliability in a distributed system, they cannot rely on having both.

To run a distributed application with reliable, consistent persistent stores, it is necessary to ensure that no references to remote objects ever become persistence reachable and to ensure that no process that uses an object remotely is long-running, in order to limit the obligation of the store providing remote access to the object.

Alternatively, to take advantage of the orthogonal persistence model in applications running over distributed persistent stores, the application programmer must make a tradeoff between the simplicity of using distributed objects that can become persistent and the consequent lack of reliability and consistency in their persistent stores.

Section 6.2 explores the issues of problem 2 above, associated with determining persistence

by reachability across a distributed system, and describes the extra support developed to help address this issue for PJRMI.

Where the orthogonal persistence model has higher priority, it is still recommended that indefinitely maintaining references between distributed, persistent stores is avoided. Section 6.3 presents support for a compromise to address problem 3 above. This compromise provides the benefit of persistent, distributed objects, but restricts it to within the lifetime of a distributed application. A conservative position is taken on the persistence of remotely-accessible objects for the duration of an application's lifetime. The compromise involves introducing time limits, appropriate to the duration of a given application's lifetime, on the remote accessibility of objects and on the usability of references to remote objects. The long-term usability of references to remote objects is traded off against the increased autonomy of persistent stores, with the intention of increasing the stores' long-term maintainability.

6.2 Determining Persistence Across a Distributed System

6.2.1 Orthogonal Persistence in a Distributed Context

PJama supports persistence by reachability from named roots of persistence. Within one PJama VM (PJVM), such reachability is determined each time a stabilisation is initiated. At stabilisation, persistent object updates are propagated to stable storage automatically. The challenge for PJRMI is to be able to determine whether an object should be made persistent because of its reachability from *remote*, persistent roots.

In theory, the rule of persistence by reachability can be applied to a distributed system as follows:

1. An object will become persistent if it is referenced by a local, persistent object.
   (It will not become persistent if it is only referenced by a local, non-persistent object.)
2. It will become persistent if it is only referenced by a persistent object in another PJVM.
3. It will not become persistent if it is only referenced by a non-persistent object in another VM.

The PJama platform addresses point one by taking care of local, persistence-reachable objects. However, the PJRMI support described in section 3 does not ensure that remotely-invokable objects do not become or remain persistent: it takes a conservative approach to their persistence precisely because of the difficulty of determining when a remotely-invokable object is reachable or no longer reachable from a remote, persistent object. This

difficulty, particularly in the face of client store shutdowns and restarts, is explored in detail in the rest of this section.

To address points two and three, Java RMI's Distributed Garbage Collection (DGC) implementation, as introduced in section 3.1, is helpful. The exportation of an object for remote use in standard Java RMI is not sufficient on its own for that object to be reachable and so exist beyond an invocation of the Java VM's garbage collector. Only weak references track the object from tables of the RMI implementation; if they are the only references to an object, it can still be garbage-collected. Once a remote reference has been established to it though, the DGC implementation ensures a strong reference is then maintained to the remotely-invokable object within its own VM; ensuring the object cannot be garbage-collected at least as long as this strong reference is maintained. Thus, the DGC implementation can be leveraged to find out which local, remotely-invokable objects are referenced from other VMs.

However, while the DGC information tells us which objects are in use by the current distributed program execution, it does not tell us what objects must persist beyond the current program execution. If a client makes a reference to a server object persistent, and then the client terminates and wishes to use that reference at some later time when it is rerun, then additional support is necessary to determine that the service is referenced from a persistent client. Thus, it is necessary to distinguish between a reference from a persistent object in another PJVM and a reference from a non-persistent object in another VM in order to determine *persistence* by reachability.

Since PJama operates in an open environment, the distributed system can be composed of both standard JVMs with no persistence support and PJVMs, running Java over persistent stores, which do have the ability to make objects persist. There are four possible permutations for the VMs involved in the two sides of an RMI call, as illustrated in figure 3.2. This adds to the complexity of determining whether an object is persistent by reachability across a distributed system, as will be illustrated in the next section.

6.2.2 Persistence with Direct and Indirect Reachability

The reachability of objects across distributed VMs is tracked by the DGC implementation, as described below. A client obtains a reference to a remotely-invokable object in another VM, initially in the form of a marshalled stub object. The DGC client implementation detects the stub object during deserialisation and, using the object identity and VM identity held in the stub, makes a `dirty` method call back to the DGC server implementation at the VM that hosts the actual remotely-invokable object. The VM hosting the remotely-invokable object now knows that this object is referenced from the client's VM.

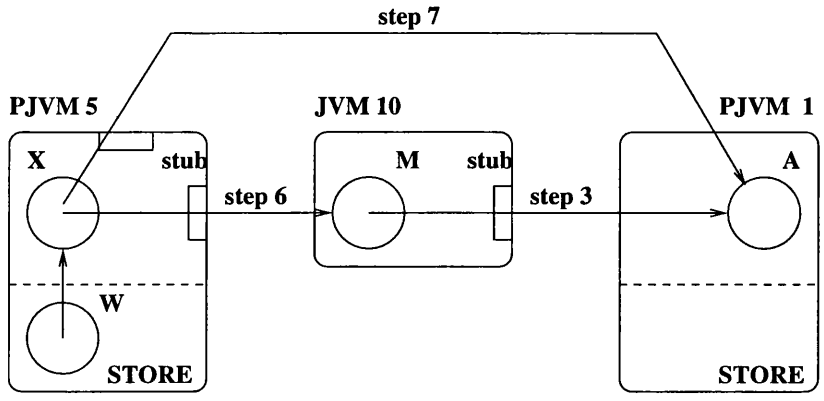

Figure 6.1: Direct and indirect reachability from a remote, persistent object

The DGC implementation is not concerned with the persistence of objects across the distributed VMs though. If the client object is made persistent then, by reachability, the referenced, remotely-invokable object should also be made persistent. A way is needed to inform the VM hosting the remotely-invokable object that this object now needs to be made persistent. In PJama, the persistence of new objects is only actually determined at stabilisation points in a persistent program. Tracing reachability from persistent objects through multiple VMs, especially in an open, persistent system where some of those VMs may have no support for persistence, raises interesting issues. These are illustrated in the steps of the following scenario (see figure 6.1):

1. A remotely-invokable object `A` is created in `PJVM 1`. It is not currently reachable from any persistent object.

2. An object `M` is created in `JVM 10`. Note that there is no support for persistence in this VM, since it is a standard JVM.

3. Object `M` obtains a reference to `A`. A stub object is created in `JVM 10`, representing `A`.

   The situation at this point is that the only reference established between VMs is the one labelled step 3 in figure 6.1. If stabilisation takes place in `PJVM 1` after step 3, then `PJVM 1` is aware that `A` is remotely-used, courtesy of the DGC tracking of remote references. `A` is not currently persistent.

4. An object `X` is created in `PJVM 5`.

5. Object `X` obtains a reference to remotely-invokable object `M`. A stub object is created in `PJVM 5`, representing `M`.

6. Object X is made persistent, by being made reachable from an existing persistent object W.

   At this point the second reference established between VMs is the one labelled step 6 in figure 6.1. After step 6, a stabilisation at PJVM 5 will make object X persistent, by reachability from W. This also means that the stub local to PJVM 5 for object M becomes persistent by reachability from X. The implication is that M and A are now also persistent by reachability but their VMs are not aware of this.

7. Object X then obtains a direct reference to A from M. A stub object is created in PJVM 5 representing A.

   The final situation for this scenario is the complete illustration in figure 6.1, where all three references are now established between the VMs. If stabilisation takes place in PJVM 1 after step 7, then PJVM 1 is aware that A is remotely-used by objects in both JVM 10 and PJVM 5. In fact, A should now also be persistent by direct reachability from X and by indirect reachability from X via object M.

The important aspects of this scenario are brought out in the paragraphs below.

Firstly, the usage of object M in this scenario demonstrates that it is possible for an object to have roles in a distributed application as both a client and a server.

Secondly, the scope of the problem of determining persistence by reachability across the distributed system can be examined using this scenario. The DGC implementation can determine reachability even when remote references are passed via intermediary sites to third party VMs. Thus, it is the DGC implementation that informs PJVM 1 when A becomes reachable from object X in PJVM 5, as illustrated in step 7 of figure 6.1. However, determining the persistence of object A at that point is a little more complex. We cannot afford to freeze the whole distributed system and do a global checkpoint that follows all references from each persistent root in the system to determine all the objects reachable from persistent roots. It's not scalable, very difficult in the face of errors and the freezing of program execution in one PJVM, because another remote PJVM wants to stabilise its objects, is not likely to be acceptable to its users; neither is the amount of time it would take to trace all the objects reachable from persistent roots across the whole distributed system.

Support could be added to PJama so that PJVM 5 can detect which local stubs have been made persistent and inform other VMs of this. After stabilisation has completed, it is possible to determine whether a stub is persistent. PJVM 5 could notify PJVM 1 when A becomes persistent by reachability. However, there is no code at JVM 10 to deal with the same sort of notification for object M. The standard JVM hosting M loads its standard JDK core classes, including those for Java RMI and DGC, locally, so there is no scope for adding extra support

here for forwarding on messages about persistence reachability. It is not clear what action should be taken on object M in this situation.

6.2.3 The Object Should Persist - But Where?

This scenario raises an interesting issue for an open persistent system. With reference to the final situation illustrated in figure 6.1, object X is now persistent by reachability from the already-persistent object W. Semantically, remotely-invokable object M is reachable from object X and should also become persistent. However, object M has been created in a JVM which itself has no support for persistence, so M cannot be made persistent locally.

Should the remotely-invokable object M be copied or moved to the site of the client object X, so that it *can* be made persistent?

Moving object M to a PJVM *with* a persistent store, such as PJVM 5 from which it is referenced, might at first glance seem a reasonable solution. However, if the scenario is extended to include other VMs that also hold references to M at this point, it quickly becomes an unworkable solution. If M was moved to PJVM 5, all references to M would have to be updated to refer to the new object at PJVM 5. The DGC tracks all the VMs that hold references to M, so identifying the VMs that have to be notified of this move would not be a problem. However, dealing with this notification would only be feasible for other PJVMs that have modified PJRMI support to deal with this. Standard JVMs with references to M may exist and have no mechanism for replacing one stub with another containing updated location information for a moved, remotely-invokable object such as M. Alternatively, if M is *copied* to PJVM 5 instead, there is no mechanism in JVM 10 for ensuring that any updates made to the original M are subsequently propagated to the copy at PJVM 5. If M has connections to a large graph of objects or it is dependent on its locality, it should probably not be moved or copied at all.

Should the autonomy of the JVM be respected?

The persistent client X will eventually get a ConnectException if it tries to use object M after the standard JDK program that created it has been terminated or fails.

A compromise.

A review of the situation reveals that, while it is not problematic to make remotely-invokable objects persistent, there are risks involved in making clients of remotely-invokable objects persistent. A compromise of referential integrity is risked by a client that is persistent or may later become persistent, when it obtains a reference to a remotely-invokable object running in a standard JVM. Since the mechanisms for obtaining a reference to a remotely-invokable

object in both JVMs and PJVMs are exactly the same, it is difficult for a client to evaluate this risk. Thus, the best recommendation is for PJRMI to track whether a client references objects in a PJVM running over a persistent store or not, and for clients to be able to query this information so they are at least better informed. If PJRMI users do not make use of this information, they must be aware that making clients of remotely-invokable objects persistent can potentially corrupt that client's persistent store.

6.2.4 PJRMI's Solution

It has been illustrated that it is a challenge to track the reachability of objects for persistence across a distributed system of VMs where some of these VMs are PJama VMs supporting persistence and others are not. This is one of the effects of supporting an open, persistent system like PJama. For PJRMI, it seems best to take a practical, conservative position when dealing with the problems raised above. This type of approach is most likely to yield working and understandable support for communication across the distributed system.

6.2.4.1 Detecting No Persistence By Reachability

Additional support is added to PJRMI for detecting where there is no persistence by reachability of remotely-invokable objects; this support builds on that provided by Java RMI's DGC implementation. In addition to the information currently collected on the references created between objects in different JVMs, PJRMI tracks which of the objects, holding references to a remotely-invokable object, are created in a PJVM running over a persistent store. Each client PJVM running over a persistent store now generates a persistent store ID. Whenever the DGC implementation detects that a VM has received a stub object, it normally sends back a `dirty` call to the VM where the stub originated, passing the VM's ID as a parameter. This allows the originating site to track which VMs hold a reference to its object. The originating site issues a lease on the reference, for which the client must regularly make renewal requests. Such requests are necessary in order to avoid the leases expiring, which could make the remotely-invokable object available for garbage collection. For PJRMI, when the DGC makes a `dirty` call for a PJVM running over a persistent store, it passes back to the originating site not only the VM ID and the lease but also the persistent store ID.

The table of information about which VMs hold references to a remotely-invokable object is made persistent at server PJVMs running over persistent stores. On store restart, this table is checked for expired leases: where a remote VM's lease has expired and it is not a PJVM that was running over a persistent store (indicated by the existence or otherwise of a recorded

persistent store ID for that VM), local remotely-invokable objects are no longer considered to be reachable from that VM. The implication of this is that if the lease has expired but this was for a PJVM that *was* running over a persistent store, then references to the local remotely-invokable objects may still be held in that store.

Thus, if a client is run in a standard JDK and it is the only client of a remotely-invokable service, that service will become unreachable after the termination of the client. Alternatively, if a client is run over a persistent store and it is the only client of a remotely-invokable service, any services it uses will become persistent, unless the client drops its reference during program execution, the reference is garbage-collected and the DGC implementation informs the server that the remote reference no longer exists.

6.2.4.2 Determining Non-persistence of Remotely-invokable Objects

Additional support is added to PJRMI for determining which objects are clients of remotely-invokable objects in VMs with no persistence support. Where a remotely-invokable object is created in a PJVM running over a persistent store, the stub object generated for it will include a persistent store ID. PJRMI determines whether a client in a persistent context references an object in a context with no persistence support by checking for the existence of a persistent store ID in the stub.

6.2.4.3 Supporting the Movement of Stores Between Hosts

The addition of persistent store IDs to stubs also contributes towards support for moving stores from one host to another. The store ID in the stub identifies the location of objects as being in a store rather than in a VM execution. The PJRMI mechanism for refreshing client's stubs on first use after store restart is used to update store location too.

The relocation of a store takes place as follows, illustrated in figure 6.2.

1. At a convenient and consistent point in program execution, the store is shut down on its old host and later it is restarted on its new host.

2. A client makes an RMI call on an object in that store.

3. The PJRMI implementation at the client attempts to make the call to the remotely-invokable object at its original host. It catches the `ConnectException` raised because this service is no longer listening for incoming calls from there.

4. The PJRMI implementation initially makes the assumption that the store has moved and delegates its call to a distributed-system-level service holding registrations of per-

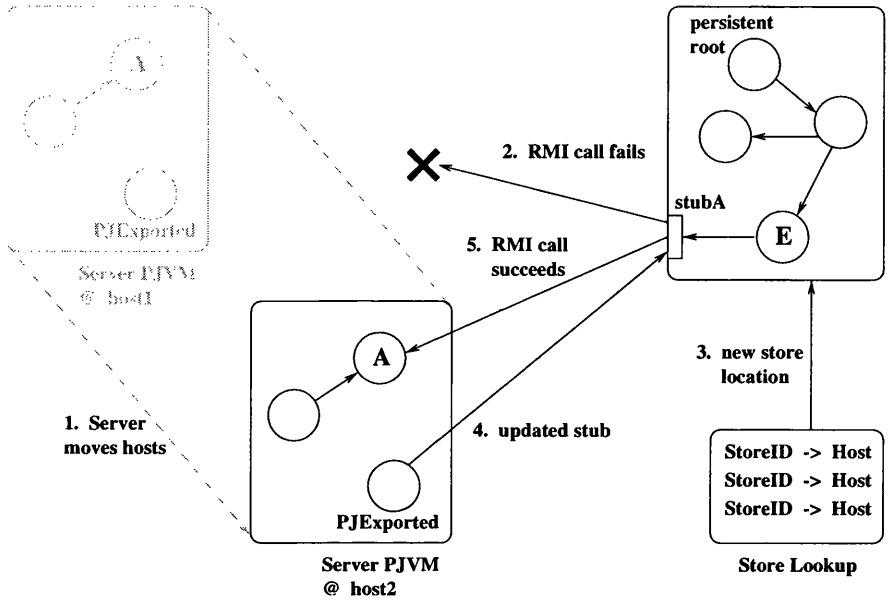

Figure 6.2: Movement of stores between hosts

sistent stores available in the system. (This uses a mechanism similar to that described in the distributed support system of the DRASTIC project [ED99]).

5. This service returns the new location of the persistent store that the client wishes to communicate with. The PJRMI implementation then uses the stub-update code supported by the `PJExported` service in the PJVM at the store's new location to obtain an up-to-date stub.

6. The client's current and subsequent RMI calls then use this new stub for the new location of the store from that point on.

6.3 Application Leases on Remote Use of Persistent Objects

With support provided for the persistence of remotely-invokable objects and of the remote references to them, consideration must be given to the maintainability of persistent stores containing such objects. It is hard to prove whether a store is maintainable over its lifetime, when that lifetime may be measured in years; such a study is outwith the scope of the author's work. However, there are steps that can be taken to improve the probability of a maintainable store.

The first step is to limit the obligations of one store by, for example, limiting remote access to objects in it. The greater the autonomy of an individual store, the greater the likelihood

that a store can maintain a consistent state over its reachable objects. After the limit on remote access to an object has run out, it can free resources that were used for supporting remote access and is free to make unreachable, locally as well as remotely, those objects that the server no longer wishes to support; they may then be garbage-collected.

The second step that can be taken for store maintainability is to limit the dependency of one store upon another. The successful execution of an application over a store is dependent on its ability to follow references between objects in order to access their state. As soon as even a single reference is established from one store to an object in another, the former store becomes dependent on the latter for the successful execution of an application that needs to follow that reference e.g. to invoke a method on the remote object.

Given that a limited persistence is desirable for connections between remotely-invokable objects and the remote references to them, support has been developed to set and maintain limits on a store's obligations and dependencies. The support for limiting a store's obligations is described in section 6.3.1, while that for limiting a store's dependencies is described in section 6.3.2.

6.3.1 Application Leases for Limiting Store Obligations

A lease, in the form of a time limit or a duration of time, can be set and enforced to limit remote access to objects to within the scope of an individual application's lifetime over a store. This application-level lease does not have the same implications as the leases used by Java RMI for distributed garbage collection (DGC).

In Java RMI, the DGC implementation uses leases and reference counting to keep track of the reachability of objects across distributed VMs, to try to ensure that a remotely-invokable object is not garbage-collected while a reference to it is still held by a remote VM. The client-side DGC implementation requests a lease on a remotely-invokable object, when it receives a reference to one, and regularly makes requests to renew the lease while the reference is still in use. This lease renewal is done automatically; the default lease is for ten minutes. The lease on the remotely-invokable object will run out if it is not renewed by a client holding a reference to it; because clients have crashed or network problems prevent clients from contacting the server. The lease will be terminated if all proxies for the object that were referenced by clients have been garbage-collected or if the clients holding such references are shut down.

Thus, if a DGC lease is granted to a client, it can only be maintained by the client as long as its process is active. The model of operation for orthogonally-persistent applications is one of maintaining the illusion of continuous operation, across shutdowns and restarts of programs running over a persistent store, as described in the OPJ specification [JA99].

Using the basic PJRMI support described in section 3, if the client runs over a persistent store, it cannot maintain the lease over a store shutdown and restart. Using the extra support described in section 6.2 relieves a client running over a persistent store from depending on regular renewal of a lease to ensure the remotely-invokable object is not garbage-collected while the client still holds a reference to it. However, it does make the client dependent on getting access to the remotely-invokable object and leaves the server with the obligation to provide remote access for at least as long as it is needed. If both the reference held by the client and the remotely-invokable object at the server are made persistent, then to try to ensure the client can always use its reference, the server is obliged to support the remotely-invokable object forever. This is because, if the persistent reference at the client becomes unreachable, it can only be garbage-collected by a disk garbage collector. It would be prohibitively costly to add support to the disk garbage collector to notify the server that the reference has been freed[1].

An application-level lease allows an application to benefit from the persistence of a remotely-invokable object during the application's lifetime, while ensuring that there is no obligation to maintain the object for remote use after that application has completed or is terminated.

6.3.1.1 Setting the Lease: Design

A lease is set on a server application process i.e. one that makes remotely-invokable objects available for remote use. Though an application process can, of course, have the role of both client and server for different objects, the significant point here is that the lease applies to its role as a server.

To set a lease on an application, the application is run within an instance of a wrapper class. This wrapper is configured with the lease during initialisation and enforced on the objects made remotely-invokable in the course of the subsequent application lifetimes.

Objects exported for remote use within the application wrapper will be remotely-invokable until the lease time limit runs out. After that time, the objects, that were exported in the course of the application's lifetime to which the lease applies, will be unexported so that they are no longer remotely-invokable. If no local references remain to the object, it is possible for it to be subsequently garbage-collected.

[1] Dave Ungar of Sun Microsystems Laboratories estimates that adding support for weak references to a disk garbage collector would increase the complexity of its implementation threefold. Tony Printezis backs this up with consideration of one of the difficulties of providing such support in his own disk GC implementation: where there is currently only one reference count per object in a store partition, two counts would be needed – one each for strong and weak references – for every object in a store partition, to support references that are weak for persistence [Pri00b].

An application lease is intended to be set with a large-grained value (i.e. with a value of hours, days or months, rather than minutes or seconds) and used with a large margin for error.

6.3.1.2 Setting the Lease: Implementation

The wrapper class `org.opj.distribution.pcopy.DistributedContext` is used for setting a lease on an application. An application programmer creates an instance of this class, referred to hereafter as the `DC`, to wrap their application. The `DC` is configured, on creation or immediately before application task invocation, with the class to be used to run the application and the lease to be enforced on its objects.

The lease is initially specified as a duration of time. Immediately before the application process is invoked, a lease time limit is set; calculated from the lease duration, where:

```
leaseLimit = currentTime + leaseDuration
```

The `DC` creates a thread to run a `DCLeaseMonitor`. This is run just before the application task is invoked, and put to sleep until the lease runs out.

The application runs as normal, invoked from within its `DC` wrapper. Once the application is running, objects can be made remotely-invokable, maintained as such and references to them can be passed to remote VMs until the lease time limit is reached. When an object is made remotely-invokable it is registered with the `DC`. The `DC` maintains only a weak reference to the object though, so that the object may be garbage-collected if it becomes otherwise unreachable.

After the lease time limit is reached, the `DCLeaseMonitor Thread` wakes up. It "unexports" all objects that were made remotely-invokable during the course of the application's lifetime in this `DC`. The unexported objects may continue to exist in their local JVM and/or store, if they are reachable locally, but they will no longer be remotely-invokable.

By iterating through the list in the current `DC`, unexporting each of the objects listed there, only the objects made remotely-invokable during the current application's lifetime are unexported, rather than removing all the remotely-invokable objects in the above tables after the current `DC`'s lease has run out.

Unexporting a remotely-invokable object removes the entries for that object from the implementation tables that track them for Java RMI: i.e. the DGC lease tables and remotely-invokable object lookup tables of the `sun.rmi.transport.ObjectTable` class. It also removes the entry for that object from the PJRMI implementation `PJamaPJExported` table. This table is used to make a persistent object remotely-invokable again on its first remote use after a persistent store restart. Removal of an object's entry from this table means that, even

if a remote JVM maintains a reference to a remotely-invokable object beyond its DC lease, it will not be possible for the remote JVM's use of that reference to trigger a re-exportation of the object for remote use, after the lease has run out.

A PJActionHandler defined for the DCLeaseMonitor ensures that its thread is recreated on a store restart. If the lease time limit has not yet been reached, the thread will be set to sleep again and remotely-invokable objects of the associated DC will be re-exported on first use. If the lease time limit has passed or is too close for any remote method invocations to be serviced before the limit is reached, then unexportation of the DC's remotely-invokable objects will take place at this time. In this case, since none of the DC's objects will yet have been re-exported for remote use, it is only necessary to remove the entry for DC's listed objects from the PJamaPJExported table, to prevent their re-exportation in the future. The DCLeaseMonitor thread then terminates.

6.3.2 Lease Management for Limiting Store Dependencies

Given the support for server-side application leases described above, once a server application process has made objects available for remote use with its application-level lease initialised, then client application processes can obtain references to these remotely-invokable objects. However, the clients can only use these references until the lease at the server runs out. The client will get an exception if they try to use those references subsequently.

Thus, the imposition of a lease on remote access to server objects also limits the usability of the references to those objects held by clients. The benefit of this is that it reduces the dependency of the client on the server; in that the client can only depend on the server for as long as its reference is valid.

6.3.2.1 Coping with a Lease: Design

The key to handling the client's now-limited dependency on remote objects is to ensure that the client does not waste time and resources making RMI calls from its reference to the server after the lease has run out and to provide informative exceptions to the client when it tries to use the defunct reference so that it knows that the timed-out lease is the reason for the failure.

To support informed client use of references to application-leased server objects, the stub objects used for these references by clients are set with the lease value of the server from which the stub originated. When a client makes a remote method call, the stub object ensures that the lease has not run out yet at the server before forwarding the call to it. If the stub finds that the lease has run out, then the call is never made to the server. Instead, an exception is

raised at the client with an informative error message containing the identity and location of the no-longer accessible object, to aid the programmer in diagnosis and handling of the failure. The tradeoff here is that the client is informed of the identity and location of the object it failed to use remotely, in order to help the client with error diagnosis, at the cost of losing location transparency and compromising security for the remote object.

6.3.2.2 Coping with a Lease: Implementation

The `DC` creates an `org.opj.distribution.context.DCLeaseServer` as a remotely-invokable object and configures it with the current lease time limit just before invoking its application, in order for a server to provide lease information to client VMs. The `DCLeaseServer` can then be contacted from the client VM's class `org.opj.distribution.context.DCLeaseClient` class in a client VM. A client uses the `DCLeaseClient` to work out the remaining lease on a server object, relative to the client's local time, and set this lease in the appropriate stubs as described below.

When a stub is serialised by the server running within a `DC`, the code of the `writeObject` method in the `java.rmi.server.RemoteStub` class ensures that:

- if the lease duration is already set in the stub, it will currently be set as a local absolute time limit: rather than serialising this absolute time limit, a duration relative to the current time is calculated from it and serialised instead, or
- if the lease duration is not set, it is left unset when serialised.

A lease duration is a period of time: five hours or thirty days, for example. A local absolute time limit is a point in time at a specific host machine: Fri Feb 18 16:46:33 GMT 2000 on the machine java.dcs.gla.ac.uk, for example.[2].

When a stub is deserialised by a client, the `java.rmi.server.RemoteStub`'s `readObject` method ensures that:

- if the lease duration is already set in the stub, it is converted to a local absolute time, or
- if the lease duration is not set, the static method `registerStub` of the `DCLeaseClient` class is called. This method adds the stub to a list of stubs, for which the lease duration should be obtained from its originating server. These lease durations are obtained after deserialisation of the current stream has finished.

[2]The implications of using lease durations and local time limits, with respect to the problems of global time in a distributed system, are discussed in section 6.3.3.2

When deserialisation of the current stream has finished, a call is made to the method `setStubLeaseLimits` of the class `org.opj.distribution.context.DCLeaseClient`, to set lease durations for those stubs that still need them. One call is made to each VM from which these stubs originated, to obtain its current DC lease time. The current time is noted just before the call is made. The lease is obtained via an RMI call to the method `getDCLeaseDuration` of class `DCLeaseServer` . The `DCLeaseServer` calculates an up-to-date lease duration, relative to the current time at the server. Back at the client, the returned lease duration is added to the time noted before the call, to obtain a local lease time limit. Then each of the stubs, in the list associated with that VM, is set with that lease limit.

During client execution, when an RMI call is made, using one of the references obtained from the server running in a `DC`, then the code of the stub class representing the reference makes a check on the lease limit held in the stub for the remotely-invokable object.

- If the lease time limit, as set in the stub, has not yet been reached then the stub goes ahead and makes the RMI call.
- If the current time is later than the limit, then a `RemoteException` is raised with an informative error message, that includes the information held by the stub on the object it represents. For example, the following error message was given on failure of a client's access to a remotely-invokable `message.service.MessageServiceImpl` object, where connections to the remotely-invokable object were originally made via port 59058 on the machine java.dcs.gla.ac.uk (represented as an IP address below).

```
org.opj.distribution.context.ExpiredLeaseException:
RemoteStub method invocation:
aborted because server's lease on corresponding object has run out:
  message.service.MessageServiceImpl_Stub
  [RemoteStub [ref: [endpoint:[130.209.240.54:59058](remote),
                     objID:[66d7b0e1:dd74619f61:-8000, 1]]]]
```

6.3.2.3 Lease at Client and Server: an Illustration

The steps taken to set a lease in a client's stub are illustrated in an example in figure 6.3. In this example, it is assumed for simplicity that the clock at both client and server are set to the same time, but this is not a requirement for use of application leases. A lease is set within a VM as a time relative to the *local* host's clock. The lease is converted to a duration for communication between the client and server VMs. The implications of using lease durations and local time limits are discussed in section 6.3.3.2. The local time limit calculated by the client will work out to be earlier than the time limit at the server, because

it takes into account the time for the lease request message to be sent across the network between client and server and the result returned. While it may cheat the client out of some time when it could interact successfully with server objects, it does ensure that the client cannot end up with a lease limit set to a later time than the limit at the server; the latter would be the case if the lease limit at the client did not take into account the time to communicate the lease duration over the network from the server.

6.3.3 Implications of Using Application Leases

6.3.3.1 Lease on Application, Not Object

A lease is set on an application process, on the basis that there is one application process running at any one time in a VM operating over one store. This is a reasonable assumption for the PJama platform, since PJama allows only one VM at a time to run in read/write mode over a store, and Java has no notion of multiple, protected address spaces within one JVM. A tradeoff is made between fine-grained control over the time limits for remote access to individual objects in a store and control over all remote access to the store for greater store autonomy. It may, for application purposes, be appropriate to provide remote access to one object in the store for a short period of time and to another object in a store for a longer period of time. However, the implication of application leases is that *all* objects made remotely-accessible in one application will continue to be remotely-accessible until the application lease time limit is reached.

More complex lease support could be provided. An application lease could be set initially on overall remote access to a store. Additional leases could then be set on individual objects made remotely-accessible during the application's lifetime; with the proviso that individual object leases can be set to run out before the overall application lease but are never allowed to be set to run out after it. The lease value in a stub would be set to: the lease on the individual object, if it exists; the application lease otherwise. The author prefers the clean-cut semantics of the current application lease though.

6.3.3.2 Leases and Time

Use of application leases does not require global clock synchronisation. This is because the lease should always be set with a large-grained value and used with a large margin for error (i.e. with a value of hours and days rather than minutes or seconds). It is also because the lease, though represented as a local absolute time limit within one VM, is converted to a duration whenever it is passed between VMs; so the validity of the lease is not dependent on the source and destination machines having clocks set to the same global time.

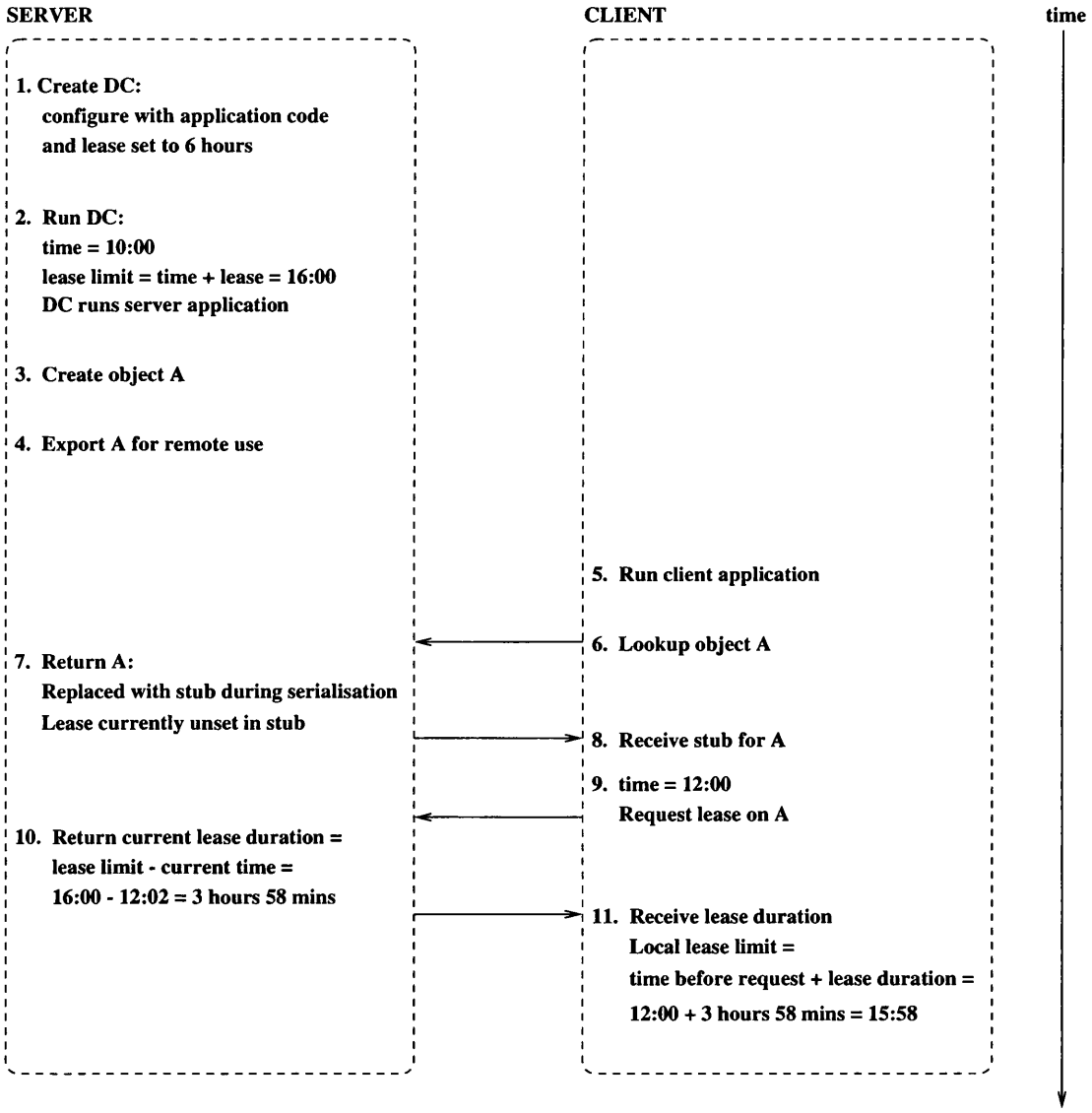

Figure 6.3: Setting a local lease limit in a client's stub

Leases could be set and used always as durations, if they were used in a non-persistent system. However, they are converted to local absolute time limits for use within one PJVM so that, if they become persistent, they should still be valid if the store containing them is shutdown and later restarted. A persistent duration cannot be interpreted correctly across a store shutdown and restart: it is impossible to determine for how long a store was shut down, so that this down-time can be deducted from a lease duration on store restart. However, a local absolute time limit is still valid on restart, since it is still comparable to the current time on the local host machine.

6.3.3.3 Reaching Lease Limit During RMI Call

There is a risk of failure for remote method calls, that are initiated by a client before the lease time limit for the object they wish to use, when the time limit is reached before the call is completed. A client may initiate an RMI call with what it considers plenty of time for the call to complete before the lease on the called object runs out. However, factors such as network delays and heavily-loaded servers can result in the lease time limit being reached before the completion of the call anyway.

The Java RMI implementation allows an application programmer to specify whether a remotely-accessible object should be forcibly unexported, even if there are pending calls or calls still in progress on the object, or whether these calls should be allowed to run/complete. For the implementation of application leases, the author has chosen to forcibly unexport remotely-accessible objects when the lease time limit is reached. While it may be debatable whether this is the correct choice for this implementation, the reason for it is that servicing all pending or in-progress calls before removing remote access effectively extends the lease to the end of execution of the last of these calls, which could be significantly later than the time limit. Imposing the time limit absolutely provides clean semantics for leases at the server, at the expense of client failures.

6.3.3.4 Telling the Difference Between Lease Limit and Server Failure

When no process is running over a store, an RMI call made by a client on an object in that store will fail. How the client handles the failure may depend on whether or not the application lease on the server-side object has run out. If the lease has run out, the non-active server cannot raise an exception to indicate this. Thus, to support an informed client, a stub contains the value of the application lease on the object it represents. If the lease has run out when a client tries to make an RMI call then, even when the server is inactive, the client gets an informative exception from the stub and it can cut its reference(s) to the stub, since the stub is no longer usable. If the lease has not run out, the client can confirm this by

checking the current value of the lease in the stub; if the server has crashed but may come back up again before the lease runs out, the stub need not be thrown away, since it's possible that it will become usable again in the future.

6.3.3.5 Server-side Persistence

Use of application leases does not compromise the consistency of a persistent store from the server's point of view. While an application lease is used to limit remote access to objects, note that it does not necessarily limit their persistence. Unexporting a remotely-invokable object to prevent further remote access to it will not confound the expectations of the programmer about the object's persistence, unless the programmer has relied *only* on an object's exportation for its persistence.

6.3.3.6 Client-side Persistence

Use of application leases does not improve the consistency of a client's persistent store. However, the application lease support provided at the client at least allows for an informative error to be raised on attempted use of a stub with an expired lease.

6.3.3.7 Non-leased Objects in a PJama VM

Because application leases are set relative to a `DC` to apply to the objects that are made remotely-invokable within it, it is still possible for objects to persist and be made remotely-invokable without a lease, when created outside of a `DC`. The implication of this is that a store makes no guarantees about remote access to objects that are not under lease: it may unexport or even drop all references to them at any time. From the client's point of view, references it holds to these objects may fail at any time and without the informative error message of a leased stub.

6.3.3.8 Interoperability Between Lease-aware PJVMs and standard JVMs

Given that PJVMs can operate in an open, persistent system where they may interact with standard JVMs, the implications of interoperability must be considered. Java RMI classes that have been extended with support for application leases are still compatible for use by standard JVMs, with regard to the rules for object serialisation and deserialisation. If a stub containing a lease is serialised and passed from a PJVM to a standard JVM, its lease will be disregarded during deserialisation. Standard JDK clients, that have obtained references to application-leased objects available from a PJama server, may attempt to make RMI calls

on those objects after the server's application lease has run out and the relevant server-side objects have been unexported.

6.3.3.9 Leasing Remote Access to a Store

Application leases are more persistently maintainable than the leases of Java RMI's DGC implementation. They are also more tailored and specific to the handling of remotely-invokable objects in a persistent system than the style of resource leases for Jini [JL99] (see section 6.3.4 for more details).

It would be hard to maintain a persistent form of DGC indefinitely across store shutdowns and restarts, because of scalability and failure problems and because stores, holding references that need to be taken into account, are not likely to always be active.

If different objects in the same store were to have different lease values, the store would not be able to do any independent store management until the last of these leases had run out, if the store is to honour the obligations implied by the leases.

If there were no leases, then a client would have no guarantee at all that it would be able to get its task done using server-side objects, before those objects disappear or at least become no longer remotely-accessible.

A lease could be applied to a VM, a transaction, a thread or an object. However, in this case it is applied to an application, running over a persistent store, that supports remote access to its objects. This is because the aim of this lease is to set one limit per store on remote access to its persistent objects.

The support for an application lease allows a client to have some confidence in getting a task done while there is sufficient time before the lease time limit. However, primarily, it provides a store with some autonomy. The lease provides one cut-off point, after which store management can be done independently of all other distributed stores.

6.3.4 Comparison with Use of Leases in Related Work

The use of application leases is similar to the use of leases [GC89] for the maintenance of file cache coherency in the V system. For this purpose, it was found that short leases on cached files were usable and fault-tolerant. However, synchronised physical clocks with bounded drift were assumed. On the grounds that this assumption is unrealistic in large-scale systems, the author of this dissertation has chosen to support application leases at a coarser grain. Application leases should rarely be affected by the clock drift of NTP, since they use lease durations to deal with different clock settings at sites distributed across a

network.

The use of leases as durations can also be found in the implementation of leases for Jini [JL99]. Unlike the implicit use of leases tied to the tracking of stubs for RMI objects in the Java RMI DGC implementation [RMI98], Jini leases are set and maintained explicitly by the application programmer on a resource. Their use is intended for dealing with partial distributed system failures and also to prevent the accumulation of resources that are no longer in use.

One of the drawbacks of Jini leases, from the point of view of persistent reference management, is that these leases are set on individual objects representing resources, rather than imposed by a context over all the relevant objects in that context. Thus, different resources in the same JVM can have radically different lease times. The intended use of application leases, in comparison, is that they ensure that all remote access granted during an application's lifetime is revoked by the same lease time limit. This leaves a store with no commitments to other sites after that lease has expired, giving the store greater autonomy.

Another weakness in Jini leases is that, typically, a client must regularly renew their lease on a resource to ensure access to it. If a client wishes to keep renewing its lease on a resource even when it is inactive, it is recommended that lease renewal is passed to a third party that does continue to be active in the meantime. The problem with this solution, in a long running persistent system, is that certain processes must be active constantly to renew leases that are to be maintained for resources that may not themselves be active for some time. The implication is that the server providing the leased service must be constantly active to grant these lease renewals and that there is some obligation on the server to maintain the leased service. This seems unrealistic for long-term maintainability, in all but the most sophisticated (and expensive) of systems that are required to stay up 24*7.

6.3.5 Future Work

6.3.5.1 Lease Extension

The support provided for application leases does not currently allow extension of those leases, but there is no reason why this could not be supported. The onus should be kept on server-control of leases, by ensuring that only the server can extend the value of its lease beyond its current time limit. The server may be able to base its decision on whether to extend a lease or not on the information on reachability to its objects from remote VMs, if it can query this information in its local DGC lease tables.

Once a lease has been extended, updating its value in the `DCLeaseServer`, it should then be a client's responsibility to find out about this lease extension. The client could check for a

lease extension on use of its stub close to or after the lease limit in the stub has expired.

6.3.5.2 Stub Lease Values From a Third Party

The current implementation of application leases allows a stub object, already set with a lease time limit on reception from its server by one client, to be passed to a third party client with the lease set as a duration based on that time limit. This means that the calculation of a lease time limit from the duration received by the third party does not take account of the communication time for the stream containing the stub when it was passed between the sending client and the receiving, third-party client.

Rather than relying on the received existing lease duration, the third-party client could instead directly contact the stub's originating server, as is the case for other stubs with no lease already set, and calculate a lease duration based on one received directly from the server. This is likely to be more accurate.

6.3.5.3 Leases Set Per Store

The current implementation sets a lease on the lifetime of an application program, on the basis that one Java VM is managed as a single address space, enforcing the model of one application task running over a VM at a time. Where multiple tasks may be supported over a single VM, as individual transactions for example, or one VM may work over multiple stores, in the future, a lease should apply to an individual *store*. This is because the aim of using leases is for increasing an individual store's autonomy.

6.3.5.4 Lease Time Limits in the face of Store Movement

Use of application leases does not require global clock synchronisation. This is because the lease should always be set with a large-grained value and used with a large margin for error (i.e. with a value of hours and days rather than minutes or seconds); and also because the lease is converted to a duration whenever it is passed between VMs. However, an absolute time lease limit within one store, set in a `DC` or in a stub object, will not be valid if the store containing it is moved from one machine to another and the two machines involved do not have reasonably-close synchronisation of their clocks. Since the lease time limit is set as a time obtained from a Java VM, it is set relative to midnight, January 1, 1970 UTC. Thus, it should be possible to consider the two machines to be synchronised sufficiently for use of lease time limits, if their clocks are synchronised at least to within a few minutes of each other. Where they are not sufficiently synchronised, extra support is needed to configure a

restarted store with the information about the difference between the clocks and to enable stubs to obtain this difference and adjust their leases appropriately.

6.3.5.5 Stub Error Handling

The problem with failure diagnosis, on raising an exception when a stub's lease has expired, is that the same object may be used by different applications. The raising of such an exception lacks the contextual information necessary to work out what application made the object remotely-accessible in the first place.

Once a stub's lease has run out, the stub will remain unusable for the rest of its lifetime at the client. A tool could be developed to replace a persistent, unusable stub with a new, usable stub, perhaps representing a different object that provides the same service as the original. Finding the stub in the store, in order to replace it, would be the first challenge. Including the stub object's persistent identifier in the failed-use error message would help here. A store maintainer could then feed this persistent identifier into the tool to identify the stub object to be replaced.

Chapter 7

Object Copying Policies: Introduction

Existing object-oriented languages and platforms used in a distributed environment typically require programmers to make decisions statically about whether objects of a particular class are passed by reference or by copy to remote sites. Where these objects are persistent, greater flexibility is required in the specification of such object passing. This is necessary to cope with the remote use of persistent objects, which have potentially large and complex object graphs, by a variety of applications and in a variety of distributed environments over the lifetime of the store.

The following chapters present distribution support integrated with orthogonal persistence for Java, providing a range of policies for deciding when object graphs are copied between widely-distributed applications running over persistent stores. Use of these policies promotes separation of architectural issues, since they can be adopted dynamically for most object classes to suit a particular application task and local or wide area network. The policies are evaluated, performance figures are given and the benefits of their use in this and other programming contexts are described.

7.1 Motivation

A number of variations on support for passing data between distributed sites have been provided over the years. Traditionally, data has been copied between sites, as typified by Birrell and Nelson's RPC [BN84] and by Argus [Lis88]. Languages including Emerald (described in more detail in section 4.3.3) have advocated "call by move" as an alternative. DCOM and, until recently, CORBA have espoused the passing of objects purely by reference. Now,

the trend for remote object access in distributed programming is moving away again from the model of passing objects solely by reference to one where objects can also be copied between processes (as in Java RMI and the Object-by-Value specification recently published for CORBA) [OH98]. Given support for both models, it is necessary for the programmer to define which should be applied to the communicated objects. For example, in a Java remote method invocation, objects may be passed as parameters or return values: these objects and all the objects reachable from them are passed by copy, unless explicitly marked (by the programmer who implemented them) to be passed by reference. Given that copying large graphs of objects between processes is expensive in terms of time and space, it is assumed that the programmer understands the implications of such copying and either only ever copies object graphs that they know are small or is willing to accept the performance costs and semantic implications of copying large ones.

Since the object-passing model is defined statically in Java on a per-class basis, this means that the manner of remote access to all instances of a given class is fixed. This seems to imply that, when designing an object for use in a distributed application, the programmer makes the assumption that it will only be used by the application for which it has been implemented and in the one context of that application's distributed environment.

Combining support for persistent objects with support for distributed applications changes these assumptions. Like traditional databases, a persistent object store is intended to be populated incrementally and maintained over months or years. This means that persistent objects may be used by different applications over time: for example, one application adds objects to the store, another browses the objects in the store while a third updates the state of those objects. The applications that will use an object later in its lifetime may not even have been envisaged when the object was first created. The persistent objects may also be used in different distributed environments over time: for example, the persistent store may be accessed over a LAN during one application lifetime while it is accessed over a WAN during a different application lifetime. As objects often build up incrementally in persistent stores over time, the stores tend to contain large object graphs. Thus, for example, an application that remotely-accesses a persistent object by making a deep copy of it may be able to do so efficiently during executions early in the lifetime of the store, but it may find the costs of such copying become prohibitive or even error-prone over time, as the object graph grows.[1]

It is *necessary* to populate a persistent store incrementally when the volume of data to be stored is too large to create objects for it and make it persistent all in one go. GAP, a Geographical Information System developed at the University of Glasgow, is a good example of

[1] It should be noted that handling the build up of large object graphs is equally applicable to OODBs and to long running systems with potentially large in-memory object graphs too.

an application that both requires storage of large volumes of data and that allows new data to be added to the store incrementally over time. This application, developed originally in Java and subsequently ported to PJama, stores mapping data. The UK Ordnance Survey data store is about 420MB, while the US TIGER data store for some of California is 1.5GB. The graph of objects reachable from one root in the former contains 699434 objects, totalling 30.45MB in size. Given the availability of US TIGER mapping data, it is possible to add new US states to an existing store as required, during the lifetime of the store.

Given the above changes in assumptions, the static, per-class definition of how objects should be remotely-accessed, as required by CORBA, DCOM and Java RMI, is not sufficiently flexible. While for some intrinsically local objects, a static definition is suitable, for the majority of objects, a more dynamic model is required; particularly given the increasing ubiquity of wide-area computation with sophisticated data usage across the Internet.

Dynamic specification of object-passing policies for remote method invocation has a number of advantages which address the changed assumptions described above. Firstly, there is a separation of architectural issues: the object-passing policy can be specified separately from a particular application's code or a particular object's class. Secondly, greater flexibility of remote object usage is supported. The benefits include adaptability of the *copying* of object graphs between processes to the scale of the network, the manner in which the object is manipulated remotely by the current application and the size of the graph of objects reachable from the accessed object.

Support for a range of dynamically-set, object-copying policies has been developed for the persistent object system OPJ. Policies have been developed to handle the copying of large object graphs in a controlled manner. Their design and implementation is presented in chapters 8 and 9 respectively, and evaluated in use with applications in chapter 10.

7.2 Assumptions

The following summary of assumptions hold for the work presented below on object-copying policies:

- large, complex object graphs build up, often incrementally, in persistent stores over time;
- some applications do require copying of object graphs between distributed processes;
- consistency of these copies, where required, is handled by the application – cache coherency is not being supported by the platform;

- object migration is not addressed;
- persistent objects of the same class may be used
  - by different applications over time and
  - in different distributed environments over time;
- static, per-class definition of object-passing between distributed processes is not sufficiently flexible for a long-lived system;

Where appropriate, these assumptions are explored in more detail in the evaluation of object-copying policies presented in section 10.

Chapter 8

Object Copying Policies: Design

Having made a case for more flexible object-passing policies and the need for policies which control the copying of object graphs between persistent stores, this section examines the drawbacks of the standard Java object-passing policies in more detail and presents the design of object-passing policies for use with PJRMI.

8.1 Object Passing in Java RMI

Java RMI is an example of distribution support which requires the programmer to make decisions, about how objects should be passed to remote sites, at the point of defining the object's class. Java RMI's support for making method calls between distributed processes can involve passing objects as parameters or return values. Java Object Serialization [JOS97] (JOS) is used to serialise (marshal) and deserialise these objects. The rules for object-passing in RMI are:

- If the object's class implements the `java.rmi.Remote` interface then the object is passed by reference. In the serialisation, the object is substituted with a stub object that holds information on the identity and location of the object it represents; it is this stub object that is actually copied to the remote site.
- Otherwise, if the object's class implements the `java.io.Externalizable` interface plus two serialisation methods called `writeExternal` and `readExternal` then these methods are called, giving the application programmer complete control over the format and content of the serialisation and deserialisation of that object and its supertypes.
- Otherwise, if the object's class implements the `java.io.Serializable` interface

plus two serialisation methods called `writeObject` and `readObject` then these methods are called and an application programmer defined serialisation and deserialisation of the object is performed. This may involve, for example, only writing out a subset of the fields or replacing field values.

- Otherwise, if the object's class only implements the `java.io.Serializable` interface then the object is passed by copy. A deep copy is made of the object and all the objects reachable from it, except where one of the other rules applies. Thus, if a `Remote` object is reachable, it will be substituted in the serialisation with a stub, rather than being deep-copied itself.

- If the object's class does not implement any of the above interfaces then the exception `java.io.NotSerializableException` will be raised; this results in the RMI call being aborted at the stub.

While the above list demonstrates the variety of approaches that can be taken, it also illustrates the complexity of defining serialisation[1]. An advantage of explicitly specifying remote object access on a per-class basis is that it is clear which object classes have been considered by the programmer for serialisation. All the classes that can be serialised implement the interface `java.io.Serializable`. All the classes that do not implement this interface cannot be serialised and may never have been considered for serialisation. The disadvantage is that such a fixed policy risks being applicable in only one environment. For those classes that have been considered for serialisation, it may be hard for an application programmer to be sure that they have made the right decision, at the time of writing the class definition, on how the object should always be communicated to remote sites. This is particularly likely to be a problem in a system where objects persist, since this increases the likelihood of multiple applications being developed to use the same classes in different ways and in different distributed environments. The change in context could be due to multiple applications that work over the same persistent objects; it is common to see a change in the way objects are used, as long-lived systems evolve over time. Different applications are likely to access the same persistent object graphs in different ways: one object-passing policy may be more appropriate for read-only access while another may be better-suited for write access. Different applications may also access different parts of the same object graph. The change in context could also be due to variations in the scale of the network over which

[1] Another contributing factor to the complexity of Java Object Serialization (JOS) is the confusion over the `transient` keyword, as described in [PAJ99]. The `transient` keyword was once used in its original intended sense by OPJ as an indicator of which fields of an object should not persist. However, JOS now overloads its meaning, for both Java RMI communication and for the JOS version of persistence, using it as an indicator of which fields of an object should not be serialised, or for which there is user-specified code for serialisation. For an article on the problems with using JOS for persistence, see [Jor99]. For a full critique of JOS, see [Eva99].

an application accesses persistent objects. The object-passing policy used over a local area network may cause problems with latency when applied within a wide area network.

The implications of object-passing policies for persistent systems should also be considered. On one hand, passing objects by reference in a distributed system of persistent stores can lead to a build up of references, and thus dependencies, between stores that can cause serious problems for long-term maintenance and autonomy of the individual stores. On the other hand, passing objects by deep copy of the transitive closure of their object graph can be expensive and even erroneous when the object to be passed has a large and complex object graph. If the large object graph has been built up incrementally in the persistent store over a period of time, the application programmer may not even be aware of its size, and therefore the implications of trying to do a deep copy from the top-level object of that graph. In the worst case, if the whole store is reachable from that object, they may unwittingly try to copy the whole store. Performance problems and errors because of buffer or memory overflow and heavy network loads are likely to result. This was found to be a common problem with previous work on supporting distribution for the DPS-algol [Wai88] and Napier88 [MCC+99] persistent systems, as reported in [Atk96, dS96].

The object-passing support developed for PJRMI attempts to address some of the problems described above with Java RMI. It focusses on adding extra support to the existing object serialisation code for more controlled copying of object graphs, while demonstrating the ability to define a policy separately from the class definitions of objects being passed in RMI calls.

8.2 Object Copying Policies Added to PJRMI

An object-passing policy influences which (and when) objects are serialised at source and deserialised at their destination, during communication between distributed processes. The previous section presented the object-passing policies applied by Java RMI. This section presents a range of object-passing policies for use with PJRMI. It demonstrates that greater separation of policy from individual application class definitions can be achieved. The intention is to support experimentation with the use of different policies over the same objects, based on the context in which those objects are to be used. Thus, a range of policies are provided to support, specifically, more flexible *copying* of object graphs between distributed VMs. The aim is to determine which policies are usable, have acceptable performance and are maintainable, and for which types of distributed application and execution environment they are appropriate. This chapter presents the object-copying policies that have been developed so far by the author and describes how to use them. It then defines the hooks which experts in serialisation could use to define their own policies.

8.2.1 Definition of a Policy

Each object-copying policy for PJRMI is represented by a Java class. This allows a user to specify the policy they want to use by giving its class name. The class must implement the interface `org.opj.distribution.pcopy.Policy`. The details of the interface are presented in chapter 9; it is sufficient here to say that implementation of its methods enables a policy to add to or override parts of the functionality of standard Java Object Serialisation used for Java RMI.

The policies with the following class names currently exist.

- `org.opj.distribution.pcopy.CopyToRefs`
- `org.opj.distribution.pcopy.CopyToSize`
- `org.opj.distribution.pcopy.CopyToDepth`
- `org.opj.distribution.pcopy.CopyByUsage`

These policies are described in detail in section 8.2.6.

8.2.2 How a Policy is Set for an Application

In order to maintain a clear separation between object-copying policy and application, a policy is specified in its own class as an implementation of the interface `org.opj.distribution.pcopy.Policy`. An application program could set the policy to be used by invoking a method of the `Policy` interface from its own setup code, in its `main` method for example. However, where more than one object-copying policy might be appropriate for separate lifetimes of the same application in different circumstances, this would require modification of the application code to change the policy. To avoid changing application code, a policy could be applied to an application program by providing a policy as a wrapper for an application's lifetime.

However, a more generalised wrapper is envisaged for distributed application execution, which could be used not only to apply an object-copying policy to the each of the application programs that cooperate in the distributed application, but also for other configuration of the current application's lifetime. Such configuration could include: checking that access is possible to the remote sites involved before beginning application execution, negotiating timeouts on remote object access, setting a consistency policy for objects shared between the processes involved and managing distribution-related errors. To support such a potential range of configuration issues requires a clean separation between the

wrapper in which the execution environment for an application is configured and each of the configuration issues themselves. Thus, the wrapper in which an application lifetime can be configured, independently from the application code itself, is provided by the class `org.opj.distribution.pcopy.DistributionContext`. A policy is decided upon for a distributed application. It is then set in a `DistributedContext` for each application process involved, as described below.

8.2.3 Setting a Policy Using a DistributedContext

The user is required to set the object-copying policy to be used at all the sites involved, before executing the code of a given distributed application. By default, if the user does not explicitly set an object-copying policy, the standard Java Object Serialisation rules apply. Once the policy has been set, it applies to all the serialisation and deserialisation of objects (i.e. all parameters and return values of RMI calls) to be passed by copy during the lifetime of the distributed application.

The policy is set during configuration of an instance of the `DistributedContext` policy support class. A `DistributedContext` instance binds an application program to an object-copying policy so that the two are always used together in that context. The policy and application are both specified by class name, the latter being the name of the class containing the `main` method for that application. Executing the application involves invocation of the `DistributedContext` wrapper which in turn runs the application with the appropriate settings for the context. Thus, for example, one `DistributedContext` instance may be configured for running an application using one object-copying policy within a local area network, while another is configured for running the same application using a different object-copying policy over a wide area network. A `DistributedContext` instance may be created and used once, or it may itself be made persistent, with a view to running the same application repeatedly in the same context. Details of the `DistributedContext` API and implementation can found in section 9.1.

8.2.4 Creation and Use of a DistributedContext

A tree traversal will be used as a simple example to illustrate the running of code in an instance of a `DistributedContext` and the effect of applying different policies.

The code for creating and running an instance of a `DistributedContext` is Java code that could be written by an application platform developer. For demonstration however, the utility class `org.opj.distribution.context.CreateAndRunDC` is used for this purpose in the explanations below.

The command-line for invoking the OPJ interpreter to run an application in an instance of a DistributedContext is as follows:

```
opj <Distributed Context setup class> <name for context instance>
    <policy classname> [<policy arguments>]
        <application main method classname> [<application arguments>]
```

Thus, to run the client program to traverse a tree using a object-copying policy referred to by name as CopyToRefs, the OPJ interpreter would be invoked as follows:

```
opj org.opj.distribution.context.CreateAndRunDC RefDCPCopyTestClient
    org.opj.distribution.pcopy.CopyToRefs
        pcrmi.client.PCopyTestClient $SERVICEHOST
```

The class org.opj.distribution.context.CreateAndRunDC contains a main method that creates an instance of a DistributedContext, hereafter referred to as a DC, passing configuration information as parameters to the constructor. The DistributedContext constructor sets the DC's name to RefDCPCopyTestClient, sets the policy it will use to that specified in the class org.opj.distribution.pcopy.CopyToRefs and sets the program to be invoked by the DC to the class pcrmi.client.PCopyTestClient that contains the main method for this application. Once the DC has been configured, CreateAndRunDC invokes the method DC.runTask passing the rest of the arguments supplied (in this case just the environment variable $SERVICEHOST that specifies the server hosting access to the tree of objects).

Configuration of a DC is separated from invocation of the application that has been associated with it. This allows a DC to be created and configured for a particular application, possibly making the DC persistent. Then the already-configured DC can be looked up by its name and invoked repeatedly with dynamically-supplied parameters.

Once the application is running, an upcall is made to the DC every time an OutputStream or InputStream is created for the purpose of serialising or deserialising RMI object parameters; while leaving creation and usage of I/O streams for other purposes unaffected[2]. This allows the policy to add to or override functionality during serialisation and/or deserialisation of the given objects. Each policy provides its own code to influence serialisation and/or deserialisation as an implementation of the interface org.opj.distribution.pcopy.Policy.

[2]The modifications made to RMI classes to provide this support are described in detail in sections 9.2.1 and 9.2.2

8.2.5 Platform Support Common to all Object Copying Policies

All the object-copying policies presented in this dissertation support some form of partial object graph copying, with different criteria for determining how much of the object graph is copied and when it is copied. Common to the implementation of all these policies is the use of a stub object `org.opj.distribution.PCopyStub`, as a substitute for application objects at the top level of the non-copied portions of the object graph. A `PCopyStub` holds the identity of the original object. The substitution of a `PCopyStub` for an application object is recorded by an implementation-level policy support service `org.opj.distribution.PCopyObjects`.

8.2.5.1 PCopyStub as Placeholder

Unlike the `java.rmi.server.RemoteStub` used to support remote references in standard Java RMI, a `PCopyStub` is not a medium for communication with the object it represents, but rather it is a placeholder for that object. After a `PCopyStub` has been passed in an RMI call and deserialised at its destination, it is put into a format that catches the first access made to it. A "residency check" made at this point detects that this is a `PCopyStub` object and that its originating site should be contacted in order to gain access to the object which it represents. The manner in which the remote object is then accessed is determined by the object-copying policy currently in force. This typically involves looking up the original object at the `PCopyObjects` service, with which it was registered when the `PCopyStub` was first generated, and copying it over to the accessing site. This manner of copying remote objects on first access is known hereafter as "remote-faulting".

8.2.5.2 Persistence of PCopyStubs

It is possible for a `PCopyStub` object to become persistent. It may not be desirable for it, as a representation of a remote object, to be persistent in the long-term, for maintenance reasons[3]. However, its persistence may at least be required for resilience of the current distributed application in the short-term.

When a `PCopyStub` has become reachable from a persistent object, it will be written to the persistent store. Its persistence will not stop it from continuing to be a placeholder for a remote object though: a subsequent access made to the persistent `PCopyStub` will still trigger a remote-fault to retrieve the appropriate object from the `PCopyStub`'s originating store. The object returned by the remote fault will become persistent, in lieu of its `PCopyStub`.

[3] See section 6 on the tradeoffs between supporting persistent references to remote objects and maintaining long-term autonomy of a store.

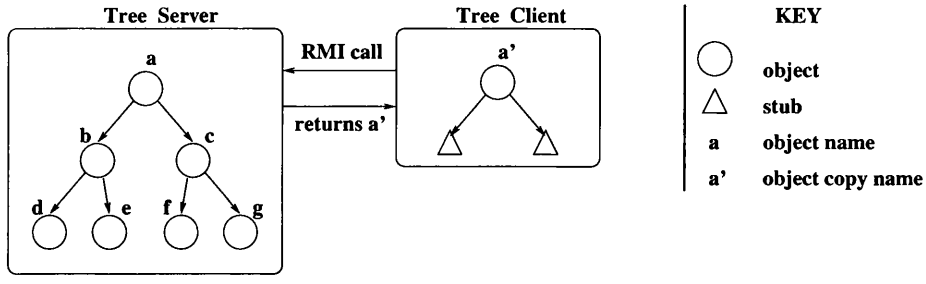

Figure 8.1: Server-side tree of objects, plus initial client-side `CopyToRefs` tree copy

Full details of the implementation supporting use of the `PCopyStub` are presented in section 9.

8.2.6 PJRMI Object Copying Policies

Specification of an object-copying policy for PJRMI involves writing methods to add to or override the functionality of the Java Object Serialisation code. Thus, if no policy is specified to override the normal Java Object Serialisation code, the standard object-passing policy for Java RMI is used. As alternatives to this, there are currently four experimental object-copying policies available. Each policy, when given an object which is not to be passed by reference, handles the object in a different way. The effect of each of these policies is described below. An evaluation of the policies is presented in chapter 10.

8.2.6.1 The `CopyToRefs` Object Copying Policy

The `org.opj.distribution.pcopy.CopyToRefs` class specifies a PJRMI object-copying policy that does an incremental copy of an object graph between sites. Initially, given an object parameter, it creates a shallow copy of the top level object to be passed to the remote site. The shallow copy contains a copy of each of the scalar field values of the given object. Each of the reference field values is replaced with a `PCopyStub` object to represent the replaced object remotely.

Thus, for example, using the tree traversal example, figure 8.1 illustrates the original tree of objects at the server. The client makes an application-level remote method call to the server to get a reference to the tree; a shallow copy of a is passed back to the client, as also illustrated in figure 8.1. Stubs have been substituted for references from object a to objects b and c.

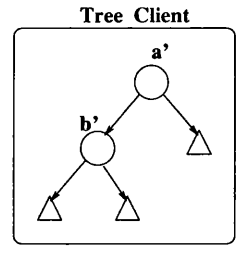

Figure 8.2: The tree copy after `CopyToRefs` access is made to b

When the client accesses a reference field of a' at the application-level, this triggers a remote-fault at the PJRMI implementation level, as described in section 8.2.5. A shallow copy of b is made from the server; all references from it are replaced by stubs. The state of the tree at the client after this call is illustrated in figure 8.2.

8.2.6.2 The `CopyToSize` and `CopyToDepth` Object Copying Policies

The policy `org.opj.distribution.pcopy.CopyToSize` makes a depth-first copy of the objects reachable from the given object; limiting the total graph size of the copied objects to below the specified size in bytes. The size of the copy depends on the parameter provided during the policy's configuration. The size is specified in bytes rather than number of objects, since objects can be of different sizes. Use of bytes is a more accurate measure of how much room the copy will take up at the client. This may be important if the client has only a small amount of memory.

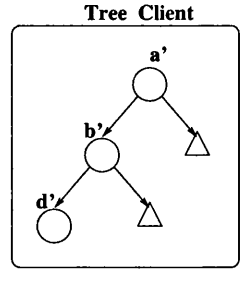

Figure 8.3: The initial depth-first `CopyToSize`

References to objects reachable from the copied object graph but outside the size limit are replaced with references to `PCopyStub` objects. Using the tree traversal example, this means that if the specified size limits copying to three objects, the client's remote method call to get a reference to the server's tree will return the graph of objects illustrated in figure 8.3.

In comparison, the policy `org.opj.distribution.pcopy.CopyToDepth` makes a breadth-first copy, that is limited to a specified depth of objects reachable from the given object graph. The depth limit is specified as a parameter during the `CopyToDepth` policy's configuration. It is specified as a number greater than or equal to one, to indicate the number of levels down the object graph to copy. As an illustration, with a specified depth of two, the tree traversal client's remote method call to get a reference to the server's tree, will return the graph of objects illustrated in figure 8.4.

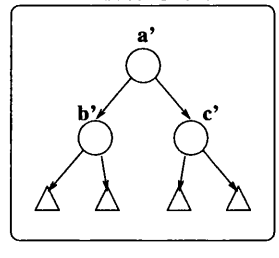

Figure 8.4: The initial width-first `CopyToSize`

Subsequent access by the client to objects `b'` and `c'` will succeed locally, while access to any other object of the graph that is currently represented as a `PCopyStub` object will trigger a further copy of the object it represents.

Consider the example of a full tree traversal, where the size limit is greater than or equal to the size of the tree, or the depth limit is greater than or equal to the depth of the tree: the number of remote calls made from client to server is reduced from a minimum of seven separate requests using `CopyToRefs` to only one request using `CopyToSize` or `CopyToDepth`.

Depending on the size of a given object graph to be passed as a parameter in an RMI call and the size or depth limit applied by these policies, a full or partial copy will be made of the object graph. The aim is to use the size or depth limit judiciously to allow full graph copying where it is of manageable size, while ensuring that copying of the full graph is prevented when it is prohibitively expensive. The latter is done on the basis that passing the limited object graph copy will normally be sufficient to provide access to the objects required.

8.2.6.3 The `CopyByUsage` Object Copying Policy

The policy specified by `org.opj.distribution.pcopy.CopyByUsage` is designed to increase the likelihood of copying the parts of an object graph that a remote application may be interested in. During the first lifetime of an application in its `DistributedContext`, this policy adopts the `CopyToRefs` policy to incrementally copy objects as they are accessed. However, as accesses to `PCopyStub` objects trigger the remote-faulting of their corresponding remote objects, the access paths traced through the graphs of the originating persistent store are recorded, indexed by the class of the object at the top level of the object graph. The recorded object graph usage information is associated with the current `DistributedContext`. Thus, the tree traversal client program's access to the server tree will initially receive the incremental copies as illustrated in figures 8.1 and 8.2.

Subsequent lifetimes of the same application in the same `DistributedContext` will make use of the usage information collected from previous runs to influence what parts of an object graph are optimistically copied (i.e. prefetched). Thus, when a top level object is passed as an RMI call parameter, the usage information is looked up and only the parts of its object graph which have been accessed in previous runs will be copied over; but this time all in one go. Where objects contain fields that have not been previously accessed remotely, the objects in those fields are still substituted with `PCopyStubs`. Thus, since in the example the client eventually traverses the whole tree in its first run, its subsequent runs against the server using this policy will receive a copy of the whole tree during the first call to the server.

Copying based on past usage is done on the basis that an application will typically follow

similar access paths through a graph of objects during repeated invocations.

8.2.6.4 Policy Evaluation

The policies described above have been implemented and used with a range of applications. An evaluation of the benefits and drawbacks of each policy, their appropriateness to applications and measurements of their performance is presented in section 10.

8.2.7 Defining New Policies

It is possible for an expert in serialisation to define their own policies, in addition to the ones that have already been provided. A summary of the hooks provided for this purpose is presented in section 9.4. It follows a description of the implementation of policy support in chapter 9, since a lot of the details of how these hooks work are presented there.

Chapter 9

Object Copying Policies: Implementation

The main aspects of the platform supporting use of policies have been introduced in chapter 8. These aspects include the use of a `Policy` interface by each defined policy, the `DistributedContext`, in which a policy is set for use with a particular application, and the `PCopyStub` used by each of the policies to represent non-copied portions of object graphs. This chapter now elaborates on each of these aspects of the platform, describing what occurs at the implementation level during policy set up and usage.

9.1 Class DistributedContext

The class `org.opj.distribution.context.DistributedContext` is used to establish what object-copying policy will be used for a particular application's lifetime. The user is required to create a `DistributedContext` at every site involved in a distributed application; ensuring that they are all configured to use the same policy [1]. When a `DistributedContext` instance is created, the following constructor is called:

```
public DistributedContext(String    DCName,
                          String    policyName,
                          String[]  policyArgs,
                          String    taskName);
```

[1] Although policy is currently set in an ad-hoc manner across all processes involved in a distributed application, a tool is envisaged for administrating such details from one site across all the processes involved, in the future.

The DCName is the name of this DistributedContext instance, used to identify this particular binding of application and object-copying policy; where a DistributedContext is made persistent, the name can be used to look it up during subsequent VM executions over the current store. The constructor registers the new DistributedContext instance under its DCName with an object in a static field of the DistributedContext class itself:

```
private static DistributedContexts distributedContexts;
```

This instance of the org.opj.distribution.context.DistributedContexts container class is registered as a root of persistence in the store.

The policyName is the fully-qualified class name of the object-copying policy to be used; for example, "org.opj.distribution.pcopy.CopyToSize". It can alternatively be set to the string "StandardRMI" to indicate that the standard Java RMI object-passing rules are to be used for executing an application in this context. Typically though, where this is required, it would not be necessary to wrap an application in a DistributedContext at all. The latter support is really only for testing and measurement purposes.

The array of policyArgs is used to configure the specified policy: for the CopyToSize policy, for example, this array would contain the size limit in bytes to which an object graph may be copied when it is first accessed. The DistributedContext constructor creates an instance of the specified policy class and initialises it, by passing the policyArgs in a call to the init method of its org.opj.distribution.pcopy.Policy interface.

The taskName is the fully-qualified name of the class containing the main method for the application to be run in this context. The DistributedContext constructor checks to ensure that this class exists.

Once an instance of a DistributedContext has been created, an execution of its associated application task may be run, either in the current VM or in a subsequent VM run over the same store. To run the application in this context, the following method of the DistributedContext class is invoked:

```
public void runTask(String[] taskArgs);
```

The array of taskArgs provides the parameters for this execution of the application.

The runTask method first calls method DistributedContext.registerDCByThread() to register the association of the current "main" java.lang.Thread with this instance of DistributedContext in a table of the DistributedContext.distributedContexts static object. This registration then supports lookup of the DistributedContext for this application by thread in the middle of application execution, to determine which object-copying policy to apply at that point; see section 9.2.1 for more details. The runTask method then invokes the main method of the DistributedContext's application task, pass-

ing the taskArgs as its parameter. This invocation is done from within a try - catch block that allows the DistributedContext to report fully on any exceptions raised.

9.2 Supporting Policy Upcalls During an Application's Lifetime

Once an application task has been invoked from within its DistributedContext wrapper object, the code executes as the application programmer intended until a call is made to serialise an object, in order to pass it as a parameter in an RMI call. At this point, hooks added to the standard Java RMI code are exercised to bring the influence of the current PJRMI object-copying policy into play.

To aid the reader's understanding of the adaptions made to serialisation and deserialisation, as described in sections 9.2.1 and 9.2.2 respectively, figure 9.1 illustrates the adapted class hierarchies. The OutputStream classes involved in serialisation are illustrated on the left, the InputStream classes involved in deserialisation are illustrated on the right and the Policy to which some method calls are redirected is shown in the middle. Method calls, made on an instance composed of this inheritance hierarchy of classes, pass from the top-most class downwards, until they reach the first definition of that method. The methods relevant for this discussion are listed for each class. The emboldened method is the one that will actually be called, overriding definitions of the same method that are lower in the inheritance hierarchy.

9.2.1 Adaption of Serialisation for Policy Hooks

The standard Java RMI code creates a sun.rmi.transport.ConnectionOutputStream to handle the serialisation of the object. In pseudo Java code, its inheritance hierarchy is as follows:

```
class sun.rmi.transport.ConnectionOutputStream
  extends class sun.rmi.server.MarshalOutputStream
    extends class java.io.ObjectOutputStream
```

Together, these classes support the object-passing policies of standard RMI, as described in section 8.1. The class java.io.ObjectOutputStream provides the standard Java Object Serialisation support, for objects whose classes implement one of the two interfaces java.io.Serializable or java.io.Externalizable; it deep-copies object graphs by default. RMI-related support is added by the sun.rmi.server.MarshalOutputStream class, for passing objects by reference if their class implements the java.rmi.Remote in-

Figure 9.1: Classes involved in object serialisation and deserialisation. (Method names not in bold type indicate a method overridden in a subtype.)

terface. The `sun.rmi.transport.ConnectionOutputStream` class supports RMI's Distributed Garbage Collector in tracking remote references.

To support PJRMI object-copying policies, a new class has been inserted into this hierarchy. In pseudo Java code, the PJRMI version of the same inheritance hierarchy is as follows:

```
class sun.rmi.transport.ConnectionOutputStream
  extends class org.opj.distribution.pcopy.PCopyOutputStream
    extends class sun.rmi.server.MarshalOutputStream
      extends class java.io.ObjectOutputStream
```

Thus, all the functionality of the standard object-passing policies is still available, while the class `org.opj.distribution.pcopy.PCopyOutputStream` provides the hooks for PJRMI object-copying policies to override certain aspects of the standard functionality where appropriate.

The class `org.opj.distribution.pcopy.PCopyOutputStream` contains the following methods:

```
public class PCopyOutputStream
  extends MarshalOutputStream
{
  ...

  public PCopyOutputStream(OutputStream out);

  protected Object replaceObject(Object obj);

  protected void outputClassFields(Object  o,
                                   Class   cl,
                                   int[]   fieldSequence);

  protected void outputArrayRefs(Object obj);

  ...
}
```

The constructor establishes which PJRMI object-copying policy is to be used during the current serialisation. It calls the method `DistributedContext.getDCByThread()`, to look up the application's `DistributedContext` and obtains the associated policy from the result.

The DistributedContext was registered with the "main" java.lang.Thread of execution on invocation of the application in DistributedContext.runTask(), so determining the main thread of execution in getDCByThread allows the DistributedContext to be obtained at this point. The main thread is obtained through method calls on the class java.lang.Thread.

The methods replaceObject and outputClassFields of PCopyOutputStream intercept calls to methods of the same name in MarshalOutputStream and ObjectOutputStream respectively. The method outputArrayRefs also overrides code of ObjectOutputStream.

To achieve this, a small number of changes were made to ObjectOutputStream. The modifier for its method outputClassFields was changed from private to protected so it could be overridden by a subclass. Also, rather than leaving the code for serialising an array of object references as part of the larger method ObjectOutputStream.outputArray, it was put into a separate method

```
protected void outputArrayRefs(Object obj);
```

which is now called from outputArray instead. This allows the code to be overridden by the method PCopyOutputStream.outputArrayRefs.

After PCopyOutputStream intercepts one of the methods that it overrides, it redirects the call to the equivalent method as implemented by the current PJRMI object-copying policy. Each of the policies implements the interface org.opj.distribution.pcopy.Policy, which includes all of the serialisation methods overridden by PCopyOutputStream, as described above.

```
public interface Policy {

  public void init(String[] args);

  public Object replaceObject(ObjectOutputStream  out,
                              Object              obj);

  public void outputClassFields(ObjectOutputStream  out,
                                Object              o,
                                Class               cl,
                                int[]               fieldSequence);

  public void outputArrayRefs(ObjectOutputStream  out,
                              Object              obj);
```

```
    public Object resolveObject(ObjectInputStream   in,
                                Object              obj);

}
```

Thus, the PCopyOutputStream makes upcalls from the standard serialisation code to the current DistributedContext to enable its object-copying policy to override the default serialisation as appropriate.

9.2.1.1 What the Serialisation Hooks Provide

The methods of the Policy interface, called from PCopyOutputStream, enable a policy to affect the serialisation as follows. The ObjectOutputStream.replaceObject method had no functionality of its own, but provided subclasses with the ability to replace the object to be serialised with a different one altogether. The MarshalOutputStream RMI class overrides this method to replace objects that implement the interface java.rmi.Remote with an instance of the class java.rmi.server.RemoteStub, to implement pass by reference semantics for the given object. The policy class PCopyOutputStream redirects calls to this method to the current policy, which can then itself maintain the functionality of MarshalOutputStream, replace it or add to it.

The original method ObjectOutputStream.outputClassFields is, in the JDK1.1.x implementation, a native method that performs the default serialisation of an object. Given a description of the type and position of each field in the object, it writes out each scalar field and then recursively invokes ObjectOutputStream.writeObject on each field that references an object or an array. The policy class PCopyOutputStream redirects calls to this method to the current policy, which can then control the recursive copying for serialisation by applying its criteria for what should be copied in its own version of this code.

The original code for serialising an array of object references, invoked from within the method ObjectOutputStream.outputArray, iterated over the array, calling the method writeObject on each object element. The policy class PCopyOutputStream redirects calls to this code to the current policy, which then controls this part of the serialisation by applying its criteria for how much of the array should be copied.

When a policy has determined that no more copying of an object graph should take place, the objects from which the rest of the graph is reachable are usually replaced with PCopyStubs. The methods described above provide the opportunities for a policy to track when replacement should occur and for this replacement to be done.

9.2.2 Adaption of Deserialisation for Policy Hooks

Similar hooks, to those used for applying PJRMI object-copying policies to serialisation, are used to allow a policy to influence deserialisation too.

When serialised objects, passed in an RMI call, are received at their destination, the standard Java RMI code creates a `sun.rmi.transport.ConnectionInputStream`, with a hierarchy of classes similar to those described in section 9.2.1, to handle deserialisation. In pseudo Java code, its inheritance hierarchy is as follows:

```
class sun.rmi.transport.ConnectionInputStream
  extends class sun.rmi.server.MarshalInputStream
    extends class java.io.ObjectInputStream
```

The class `java.io.ObjectInputStream` provides the standard Java Object Deserialisation support; by default, it recreates the state of an object graph as it was at the point of serialisation. If the object's class contains methods of the `java.io.Serializable` or `java.io.Externalizable` interface that describe a more specialised deserialisation, they apply instead. When a `RemoteStub` representing a remote reference to an object is to be deserialised, the class `sun.rmi.server.MarshalInputStream` supports, where necessary, the loading of the stub's class from a remote WWW server where those classes have been made available. The class `sun.rmi.transport.ConnectionInputStream` does its part to support RMI's Distributed Garbage Collector in tracking remote references.

To support PJRMI object-copying policies, a new class has been inserted into this hierarchy. In pseudo Java code, the PJRMI version of the same inheritance hierarchy is as follows:

```
class sun.rmi.transport.ConnectionInputStream
  extends class org.opj.distribution.pcopy.PCopyInputStream
    extends class sun.rmi.server.MarshalInputStream
      extends class java.io.ObjectInputStream
```

The class `org.opj.distribution.pcopy.PCopyInputStream` provides the hooks for PJRMI object-copying policies to override certain aspects of the standard deserialisation functionality where appropriate. It contains the following methods:

```
public class PCopyInputStream
  extends MarshalInputStream
{
  ...

  public PCopyInputStream(InputStream in);

  protected Object resolveObject(Object obj);

  ...
}
```

Similarly to `PCopyOutputStream`, the `PCopyInputStream` constructor establishes which object-copying policy is to be used during the current deserialisation. It does this via a lookup of the current `DistributedContext` using the "main" `Thread` as the key, as described in section 9.2.1.

The method `resolveObject` of `PCopyInputStream` intercepts calls to the method of the same name in `MarshalInputStream`. The call is then redirected to the equivalent method as implemented by the current object-copying policy. This gives the policy an opportunity to replace or modify the object that has just been deserialised, if required.

9.3 Policy Use of Stub Objects

As well as providing hooks for calling policy methods from serialisation code, the common support for policies also includes the class `org.opj.distribution.pcopy.PCopyStub`. During serialisation for an PJRMI call, all of the object-copying policies, that have been defined for PJRMI, copy an object graph in a limited manner, based on the copying criteria of the particular policy. In each case, the objects heading the non-copied parts of an object graph are replaced with objects that can represent that non-copied portion of the object graph remotely. The replacement objects used by each policy are instances of the `PCopyStub` class. After the copied parts of an object graph have reached their destination and they've been deserialised and traversed by application code, it is then part of the policy to define what happens when an access is attempted to a non-copied portion of the graph now represented by a `PCopyStub`.

During serialisation, to replace an object with a `PCopyStub`, a policy makes use of the class `org.opj.distribution.pcopy.PCopyObjects`. A policy passes the object to be replaced

as a parameter of the PCopyObjects method registerObject:

```
static protected Object registerObject(Object o);
```

This method creates an object identity to uniquely identify the object in the current VM, creates a PCopyStub instance to hold this identity and stores the association between the original object and its corresponding PCopyStub in a table of the PCopyObjects class. The PCopyStub is returned to the policy as the result of the method call. The policy then modifies the object field that held a reference to the original object so that it now references the PCopyStub instead. This PCopyStub is then serialised as part of the object graph and passed by copy to the destination of the current RMI call.

During deserialisation, after a PCopyStub object has been deserialised by the method of the class ObjectInputStream for reading in an object:

```
private native void inputClassFields(Object  o,
                                     Class   cl,
                                     int[]   fieldSequence);
```

an addition to this method for PJRMI object-copying support of PCopyStubs goes on to make a call to the PCopyStub method:

```
private static native PCopyStub setToProxyType(PCopyStub pcs);
```

This method takes the newly-deserialised PCopyStub object, in its normal object format, and returns it in a format that will trigger a "fault" during a residency check. In PJama, every object undergoes a residency check before it is accessed, to ensure that the object is in memory and, if not, then bring it into memory from its persistent state on disk, which is known as "faulting" the object. This residency check mechanism has been extended for object-copying support to trigger remote faulting between one PJVM and another, as well as local faulting between a store and the memory of the PJVM running over it.

9.3.1 Triggering Access to a Remote Object

In the JVM Classic implementation, every object is accessed via a handle object. The handle object contains two fields which, in normal object format, contain a pointer to the object itself and a pointer to the table of methods for that object's class. This is illustrated as the key to figure 9.2. An object is usually accessed using the JVM macro unhand that, given a handle, returns the pointer to the object. The PJama implementation has redefined this macro so that, given a handle, it uses the isAFaultBlock macro to check whether its object pointer field contains a valid memory address for a memory-resident object or whether it contains a persistent object identifier (PID) that represents a persistent object still on disk.

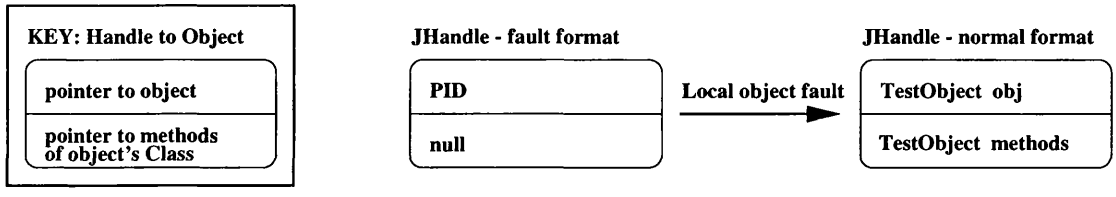

Figure 9.2: Object fault from store to VM memory

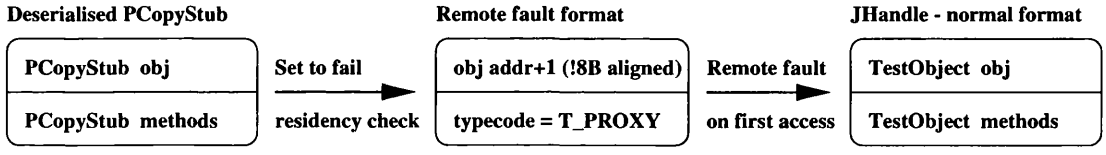

Figure 9.3: Object fault from remote VM to local VM

If the field contains a PID, an object fault is triggered to bring the corresponding object on disk into memory[2]. The change in contents of the handle itself during the object fault is illustrated in figure 9.2.

For the implementation of object-copying support, the unhand macro has been further extended so that if, on access, an object is found to be non-resident, it may be faulted from a remote VM or from the local disk. As mentioned above, a `PCopyStub` is put into a format after deserialisation that mimics a non-resident object. The transition of a `PCopyStub` object's handle, from its state on deserialisation to the "remote-fault format" state designed to fail the first stage of a residency check, is illustrated in figure 9.3. Since all object addresses are eight-byte aligned in the JVM implementation, the intentional adjustment of the `PCopyStub`'s object address, so it is no longer eight-byte aligned, ensures that the residency check fails[3]. A check added within the unhand macro then recognises that the type code in the methods field of the handle is set to `T_PROXY`: this indicates that this is really a `PCopyStub` object in remote-fault format. The `PCopyStub` handle is converted back to the normal resident object format it had when first deserialised and the code is run for providing access to the remote object that is represented by this `PCopyStub`. This usually results in the original object being copied over and the `PCopyStub`'s handle being converted to the handle of the remote-faulted object, as illustrated in the latter stage of figure 9.3.

[2]For more details of this PJama implementation, see [DA97].

[3]See section 9.3.3 for an explanation of how this format is handled during garbage collection

9.3.2 Accessing a Remote Object

When access is attempted to a remote object that is locally represented by a `PCopyStub`, the access is caught by a residency check as described above; at that point the information held in the `PCopyStub` is used to determine the location and identity of the remote object. The identity is held as an `ObjID`. It was registered, along with the object, in a table of the `PCopyObjects` class in the PJVM from which the `PCopyStub` originated. The location is held as a reference to a remote lookup service. The `PCopyObjects` class also supports this remotely-invokable, PJRMI implementation-level service through the interface `org.opj.distribution.pcopy.PCopyObjectService`.

Once a residency check has revealed a `PCopyStub`, its method `getRealObject` makes an RMI call to the following method of the `PCopyObjectService`, passing the object identity as the key in the remote lookup:

```
public Object getObjectCopy(ObjID id);
```

This method returns the object on which the access, that triggered the residency check, should now go ahead. The returned object is typically a copy of the original object's graph, but may only be a partial copy, since the current policy will be applied to the result of this call too.

9.3.3 PCopyStubs and Garbage Collection

Setting the remote fault format of a handle's pointer to a PCopyStub object involves incrementing its address by one so it is no longer eight-byte aligned. This means that as long as the handle is in remote fault format, the referenced PCopyStub object is not actually reachable and could be considered garbage erroneously. To avoid this, `PCopyStub` handles are reset from remote fault format to normal object format for the duration of the garbage collection (GC). No remote faulting should ever be triggered during a GC anyway: an exception is raised if this occurs. Remote fault format of still-reachable `PCopyStub` handles then is reestablished at the conclusion of the GC, before application program execution continues.

9.3.4 Persistence of PCopyStubs

If a `PCopyStub` becomes reachable from a persistent object, it will be promoted from the Java heap to the PJama object cache and written to the persistent store. This will occur either within an explicit call to stabilise persistent state in the middle of program execution or implicitly at the successful completion of a program execution. Because a `PCopyStub` is set in remote-faulting format after deserialisation at its destination, this must be recognised

and changed back to normal object format before its promotion can proceed.

The handle to a PCopyStub will be in remote faulting format when the promotion code comes across it: this format is the one illustrated in the middle of figure 9.3. The format is recognised from the type code set to T_PROXY in the methods field of the handle. A macro is called to re-instate the normal object format for a PCopyStub object, as illustrated in the first stage of figure 9.3. The PCopyStub object can then be promoted like any other normal object.

Subsequently, as for any other persistent, non-resident object, an access attempted on a PCopyStub that's still on disk will initially result in a local residency check, which will trigger the faulting-in of the object from disk to object cache. Once the PCopyStub is resident in memory, the second phase of the residency check will recognise this object as a placeholder for a remote object and trigger a remote-fault to retrieve the appropriate object from the PCopyStub's originating store. The object returned by this remote-fault will become persistent by reachability, because its PCopyStub was reachable, and all future accesses to the PCopyStub will be redirected to the newly-faulted object it represents.

A complete illustration of the formats of a PCopyStub handle at various points in its lifetime is provided in appendix C.1.

It should be noted that the remote-faulting of an object, triggered on access to its PCopyStub, can only succeed if the original object still exists in the store from which the PCopyStub originated, and only if a server process is currently up and running over that store and is accessible over the network. See section 6 for an exploration of the issues associated with extending persistence by reachability across a distributed system to help ensure that a remotely-referenced object persists as long as it is needed.

9.4 Hooks for New Policies

To summarise the support provided for object-copying policies and to make it clear what a serialisation expert needs to do in order to implement their own policy, the hooks for policy support are reviewed in this section.

9.4.1 How to Implement the Policy Interface

The API for policy support is provided by the org.opj.distribution.pcopy.Policy interface. This interface is defined as follows:

```
public interface Policy {

  public void init(String[] args);

  public Object replaceObject(ObjectOutputStream  out,
                              Object              obj);

  public void outputClassFields(ObjectOutputStream  out,
                                Object              o,
                                Class               cl,
                                int[]               fieldSequence);

  public void outputArrayRefs(ObjectOutputStream  out,
                              Object              obj);

  public Object resolveObject(ObjectInputStream   in,
                              Object              obj);

}
```

The main class for defining a specific policy must implement this interface. The purpose of the individual methods of this interface are summarised below. These are the steps that must be followed to provide a `Policy` implementation.

9.4.1.1 Step 1: initialise policy on DistributedContext creation

Write the `init` method to initialise fields of the policy implementation, before first use of the policy for serialisation. It will be called from the constructor of a `DistributedContext`, passing it the policy arguments supplied as one of the constructor's parameters. This supports initialisation of a policy before application execution begins.

9.4.1.2 Step 2: use serialisation methods to restrict copying

Override the methods `replaceObject`, `outputClassFields` and/or `outputArrayRefs` with code to achieve the desired effect of your policy during serialisation.

This work is likely to fall into two parts:

1. tracking the serialisation of an object graph, to determine when the criteria are met

for curtailing the copying of an object graph for serialisation and

2. substituting non-copied objects with instances of the class `PCopyStub`, so that the stubs are serialised instead of the rest of the object graph.

To do the latter, calls will need to be made to the following method of the `PCopyObjects` service:

```
static protected Object registerObject(Object o);
```

which takes the object to be replaced as an argument and returns the `PCopyStub` containing the corresponding object identity as the result. It is then the responsibility of the policy to place the returned `PCopyStub` at the appropriate place in the serialisation of the object graph, so that it does actually replace the object for which it is a substitute.

9.4.1.3 Step 3: use de-serialisation method

If any adjustment to the serialised objects is necessary, after deserialisation at their destination and before they are made accessible to the user, this should be done within the `Policy.resolveObject` method.

9.4.2 Leaving the Rest to the Policy Support

Once the policy has been defined, the policy support will take care of ensuring that the appropriate policy methods are called during serialisation and deserialisation. Once object graphs containing `PCopyStub` objects have been received at their destination, the policy support handles the mapping of an access made to a `PCopyStub` to an access on the object it actually represents.

9.5 Implementation of Individual Object Copying Policies

Having summarised the policy support and how it is used to implement a policy, the implementation of the pre-defined policies introduced in chapter 8 is now described. This will illustrate how the policy support is used in practice and show how the implementations of each of the policies differ.

9.5.1 Behaviour Common to the Policies

The effect of each policy may be felt during:

1. initialisation of the policy,
2. serialisation of an object for an RMI call and
3. deserialisation of an object for an RMI call.

In the serialisation of objects for RMI using each of the policies presented below, objects defined to be passed by reference to remote sites are still passed by reference. Thus, if an object implements the `java.rmi.Remote` interface, it is replaced by a corresponding `java.rmi.server.RemoteStub`, as in the default method `replaceObject` of the class `sun.rmi.server.MarshalOutputStream`. This includes the substitution of a reference to the `PCopyObjects` service with a `RemoteStub`, in a `PCopyStub` object itself.

9.5.2 Policy CopyToRefs

The policy defined in the class `org.opj.distribution.pcopy.CopyToRefs` supports incremental copying of object graphs on remote access, where each object in the graph is shallow-copied and its references to other objects are replaced with `PCopyStub` objects. Access to the stubs subsequently triggers remote faulting of the corresponding object. The effect of this policy is described in section 8.2.6.1.

9.5.2.1 CopyToRefs Initialisation

No initialisation of the `CopyToRefs` policy is required before application execution begins.

9.5.2.2 Serialisation with CopyToRefs

The serialisation code is modified, for objects passed by copy, by this policy's implementation of the methods `replaceObject` and `outputClassFields`, as follows:

- For objects defined to be passed by copy, the top-level object's fields containing references to objects are replaced with references to `PCopyStubs` before the top-level object is serialised.
- If the top-level object is an array of references to objects, it is replaced with an array of references to `PCopyStubs`. To ensure the array is now serialised as an array of `PCopyStubs`, its real type is moved to a temporary variable in the first `PCopyStub` element of the array and the type of the array is changed to be an array of `PCopyStubs`. Alternatively, if it is an array of scalar elements, the whole array is serialised.

9.5.2.3 Deserialisation with CopyToRefs

An array of objects, that has been replaced with `PCopyStubs`, is serialised and deserialised as an array of `PCopyStubs`. However, to avoid type-checking problems during subsequent usage at the destination after deserialisation, the original type of the array is reinstated in a call to the policy's definition of `resolveObject`.

9.5.3 Policy CopyToSize

The policy defined in the class `org.opj.distribution.pcopy.CopyToSize` supports depth-first copying of an object graph to a specified size limit in bytes. References to parts of the graph still to be copied, when the size limit is reached during serialisation, are replaced with `PCopyStubs`. The effect of this policy is described in section 8.2.6.2.

9.5.3.1 CopyToSize Initialisation

This policy is configured with a call to the `init` method, that passes one argument: the object graph size limit in bytes for this application's lifetime. After being set from a `DistributedContext` constructor, it will apply to all object graphs passed by copy during the lifetime of the associated application.

9.5.3.2 Serialisation with CopyToSize

The serialisation code is modified, for objects passed by copy, by this policy's implementation of the methods `replaceObject` and `outputClassFields`. For each object to be serialised:

- the object's size is calculated (see below for details),
- if the total size of the graph serialised so far plus this object's size equals less than the size limit set during policy configuration, the object is serialised and the total size adjusted accordingly,
- otherwise, once the maximum graph size has been reached, then all references to objects are replaced with `PCopyStubs`.

An object's size is determined from the instance size held in the class at the VM implementation level: i.e. it is the size of the object memory referenced from the handle's `obj` pointer (see the key in figure 9.2 for an illustration of a handle to an object). It does not take into

account the memory used for the handle object itself or the class and methodtable objects that are also referenced from the handle, since neither of these are directly serialised so don't vary the size of the serialisation. Similarly, the size of an array is calculated to be the size of one of its elements multiplied by the length of the array.

Since an object graph is serialised using recursive calls to the `writeObject` method of the class `ObjectOutputStream`, a `recursionDepth` attribute of class `ObjectOutputStream` is used to track when serialisation has finished serialising one object graph and is starting on a new one. The `recursionDepth` has a value of one when at the top-level of an object graph; it is incremented on each subsequent, recursive call to the `writeObject` method and decremented on exit from the same call to keep track of the object graph's current depth. In standard serialisation, this is used in tracking the beginning and end of the writing of a particular class of object so they can be marked in the serialisation. This is useful, for example, where one version of the class is written and a different version is read, since unexpected fields of a new version may be skipped by an older version. Similarly, the `recursionDepth` is also used by this `CopyToSize` policy to determine when it has finished tracking the size of the previous object graph and is now tracking the size of a new object graph being serialised.

9.5.3.3 Deserialisation with CopyToSize

No code, additional to the default deserialisation, is required for this policy.

9.5.4 Policy CopyToDepth

The policy defined in the class `org.opj.distribution.pcopy.CopyToDepth` supports breadth-first copying of an object graph to a specified depth, where one indicates only the top level object, two indicates the top level object and all those only immediately reachable from it, etc. References to parts of the graph still to be copied, when the depth limit is reached during serialisation, are replaced with `PCopyStubs`. The effect of this policy is described in section 8.2.6.2.

9.5.4.1 CopyToDepth Initialisation

This policy is configured with a call to the `init` method, that passes one argument: the limit on the depth of the object graph for this application's lifetime. After being set during creation of a `DistributedContext`, it will apply to all object graphs passed by copy during the lifetime of the associated application.

9.5.4.2 Serialisation with CopyToDepth

The serialisation code is modified, for objects passed by copy, by this policy's implementation of the methods `replaceObject` and `outputClassFields`. For each object to be serialised:

- While the `ObjectOutputStream.recursionDepth` is less than the depth limit specified during policy configuration, continue to serialise the objects of the current object graph.
- If the `ObjectOutputStream.recursionDepth` becomes greater than the policy's current depth limit, replace all references to objects with references to corresponding `PCopyStubs`.

9.5.4.3 Deserialisation with CopyToDepth

No code, additional to the default deserialisation, is required for this policy.

9.5.5 Policy CopyByUsage

The policy defined in the class `org.opj.distribution.pcopy.CopyByUsage`, and its associated helper classes, supports the copying of an object graph based on past usage by the current application. Keyed on the class of the top-level object, the access paths made through a given object graph are tracked during application's lifetime. Subsequent use of objects of the same type results in the copying of objects in the graph that have been previously accessed. The effect of this policy is described in section 8.2.6.3.

9.5.5.1 CopyByUsage Initialisation

This policy is configured with a call to the `init` method of `CopyByUsage`, which creates an instance of this policy's helper class `org.opj.distribution.pcopy.TrackUsage`. A single `TrackUsage` instance is created for use by a specific `DistributedContext`. It may be used for one or for repeated lifetimes of the same application task in the same `DistributedContext`. Thus, after being created as the result of this method call from the `DistributedContext` constructor, it will be used to track the usage of all object graphs passed by copy during the lifetime of the associated application. The cost of tracking usage is most likely to be amortised if the same application is executed repeatedly from the same `DistributedContext`, taking advantage of the usage information collected in the `TrackUsage` tables during previous runs.

9.5.5.2 Serialisation with CopyByUsage

As with all the other policies described in this chapter, objects defined to be passed by reference to remote sites are still passed by reference. Thus, if an object implements the `java.rmi.Remote` interface, it is replaced with a corresponding instance of the class `java.rmi.server.RemoteStub`, as in the default method `replaceObject` of the class `sun.rmi.server.MarshalOutputStream`. This includes the substitution of a reference to the `PCopyObjects` service with an instance of a `RemoteStub`, in a `PCopyStub` object itself.

The serialisation code is modified, for objects passed by copy, by this policy's implementation of the methods `replaceObject` and `outputClassFields`. They make calls on the policy's support class `TrackUsage` to establish and maintain information on what classes are copied and used remotely.

Serialisation: first class use

Initially, the first instance of a class, that is passed by copy, is serialised as a shallow copy of the top-level object, with its references to other objects replaced with `PCopyStubs` (as in the `CopyToRefs` policy, first described in section 8.2.6.1).

The `TrackUsage` object associated with this policy contains the field:

```
private Hashtable classUsageTable;
```

which is used to hold collected information on accesses made to objects, using the class of the top-level object of a serialised object graph as the lookup key. It also contains the field:

```
private Hashtable objectLookupTable;
```

which is used to map the object identities, held in `PCopyStubs`, back to the original objects that they represent in a serialised object graph. These two `TrackUsage` tables are illustrated in figure 9.4, for reference during the following explanation of how they are used.

The first instance of a class to be serialised initialises tracking of the class:

- An entry is created for that class in the `classUsageTable`: registered in the table using the class name as the key, this `ClassUsage` entry keeps a record of the class's non-static reference fields: those that would appear in an instance of the class.
- The `ClassUsage` entry is initialised with an array of `FieldEntry` objects, one per reference field of the class.
- The `FieldEntry` contains
  - the reference field name from the class,

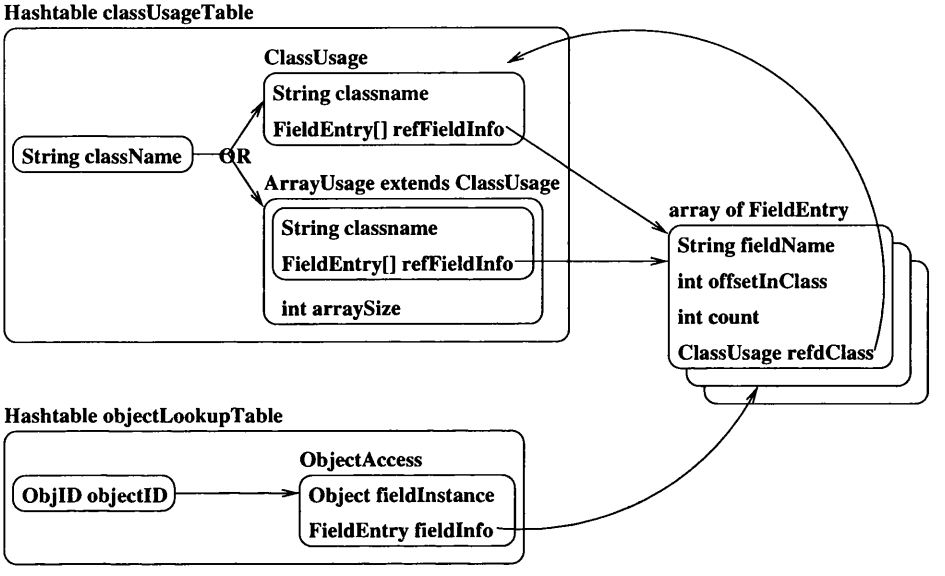

Figure 9.4: class `TrackUsage` tables of the `CopyByUsage` policy

  - the offset of this field within an instance of the class,
  - a count of the number of times this field has been accessed from an instance of this class - which is initialised to zero, and
  - a reference to the `ClassUsage` for the class of the field itself.

- If the top-level object is an instance of an array of objects instead, an entry is created for the array in the `classUsageTable`: registered in the table using the array class name as the key, this `ArrayUsage` (an extension of `ClassUsage`) keeps a record of the reference fields of the array. It also records the size of the array.

- Each reference field of the top-level object or array is replaced with a `PCopyStub`. Unlike the other policies, the service referenced from the `PCopyStub` is not the implementation of `PCopyObjectService` supported by an instance of the `PCopyObjects` class. In this case, the `TrackUsage` instance provides the implementation for the `PCopyObjectService` instead; so it is a reference to the `TrackUsage` instance that is put into the service field of the `PCopyStub` for this policy.

- An entry is created for each replaced field in the `objectLookupTable`. This entry is registered using the object identity `ObjID`, that has been generated for the field's `PCopyStub`. The entry, of class `ObjectAccess`, contains

  - a reference to the object originally held in the field, before substitution with a `PCopyStub`, and

– the FieldEntry for the field, that holds the information already described above, including the offset of this field in the top-level object that contains it.

Serialisation: mid-graph objects copied on access

After the top-level of an object graph has been passed by copy to its destination, access to one of its reference fields, currently containing a PCopyStub, triggers a remote-fault, as previously illustrated in figure 9.3. Since the service field of the PCopyStub holds a reference to the originating site's TrackUsage, as described above, it is this object that is contacted with the remote method call on PCopyObjectService.getObjectCopy.

The TrackUsage implementation of the getObjectCopy method is passed the PCopyStub's ObjID: it uses this object identity to do a lookup on the objectLookupTable, which returns the corresponding ObjectAccess.

The ObjectAccess contains the object to be returned to the accessing site and a FieldEntry: information on the field holding this object. To track the fact that the object is now being accessed, a call is made to increment the FieldEntry's access count. To serialise the object to be returned, the FieldEntry.refdClass, containing the ClassUsage information for the class of object, is used. If no usage information for the class of this object yet exists, a shallow copy of this object is made and PCopyStubs are substituted for the other objects referenced from this one. Alternatively, if usage information does already exist, the object is serialised as described below.

Serialisation: object graphs of previously-tracked classes

Once usage information is held on a class, it can be applied to subsequent serialisations of instances of that class, done in the same DistributedContext, either in the same or in subsequent application lifetimes. If an application object is passed as a RMI parameter, a lookup on its classname in the classUsageTable returns its ClassUsage. If an access has been made to a PCopyStub at a remote site, the ClassUsage for the object to be returned to that site is obtained via an ObjectAccess as described above.

Where previous usage information is held in the ClassUsage, this takes the form of non-zero count values in the FieldEntrys associated with fields of the class. For the object to be serialised, each of its top-level primitive fields are serialised first. Then, for each reference field of the object's class,

- if its FieldEntry contains a count value of zero, that field has not previously been accessed remotely, so the object in that field is substituted with a PCopyStub during serialisation, otherwise
- if its FieldEntry contains a non-zero count value, the primitive fields of the object

in that field will also be serialised, and then

- the `ClassUsage` for the object in that field will be retrieved from the `refdClass` attribute of the `FieldEntry` and, for each of its `FieldEntrys`, the same rules are applied, recursively.

The result is a serialisation of all the fields of an object graph that have previously been accessed, with `PCopyStubs` replacing objects in fields not previously accessed. This prevents previously unused portions of object graphs for a particular class, and those reachable from it, being copied over to a remote site.

9.5.5.3 Deserialisation with CopyByUsage

No code, additional to the default deserialisation, is required for this policy.

Chapter 10

Object Copying Policies: Evaluation

10.1 Introduction

The motivation for object-copying policies, as presented in section 7.1 can be summarised as follows:

- large, complex graphs of objects build up incrementally over time in persistent stores;
- copying the full transitive closure of a large object graph between processes participating in a distributed application can be prohibitively expensive in terms of time and space;
- persistent objects may be used by different applications over time;
- persistent objects may be used in different distributed environments over time and
- per-class static definition of the object passing policy for an object is not sufficiently flexible for handling the problems above for distributed, persistent objects.

The support for object-copying policies, as presented in sections 8 and 9, addresses these points. Firstly, there is a separation of architectural issues: the object-copying policy can be specified separately from a particular application's code or a particular object's class. Secondly, greater flexibility of remote object usage is supported. The evaluation presented below demonstrates this flexibility by examining the object-copying requirements of some distributed applications and describing the effects of applying object-copying policies; with measurements for illustration where appropriate.

10.2 Separation of Architectural Issues

Java RMI is an example of the type of system that requires a *static* definition to indicate whether or not an object is to be passed by reference. Other existing work, including CORBA's Value Type Semantics (see section 4.2.6) and FlexiNet (see section 4.2.11), take care to avoid such static definition of object passing policy on the object's type itself. The support for object-copying policies described in this dissertation takes the latter approach in order to achieve the flexibility in the handling of persistent object graphs that is likely to be needed through their lifetime.

Thus, object-copying policies are specified independently of the classes of objects used by applications. This promotes separation of policy from class definition, enabling a policy to be applied on a per-application-lifetime basis. The intention is not to be able to change the way one application accesses an object remotely where this is inappropriate. Changing the object-copying policy for a specific application's use of an object may violate assumptions made by the application about, for example, the consistency of the application's view of the object with its state at its original site. However, the intention is that the support for applying policies on a per-application-lifetime basis does allow different applications to influence communication of the same object graph in different ways, where this is deemed appropriate by the application programmer. The object-copying policies that have been defined provide the ability for applications to influence communication of object graphs specifically with regard to control of copying between sites.

Sections 8 and 9 have described the design and implementation of support for definition of a policy in its own class, and for specifying and applying a particular policy to the lifetime of a distributed application. This clearly demonstrates the required separation of concerns. The use of object-copying policies does enable greater control over the copying of object graphs between sites in a distributed application lifetime than previously supported for Java RMI. The sections below illustrate this by applying a number of policies to some distributed applications.

10.3 Measurements Setup

A test environment has been created for taking measurements on the execution times of a number of distributed applications using a range of object-copying policies. The aim of taking these measurements is to determine the cost of applying various object-copying policies to an application. This contributes to an evaluation of the use of such policies, based not only on the execution times of an application but also on its usability and reliability of access to the data that it uses.

Measurements have been taken on the use of object-copying policies over local and wide area networks. Where communication is presented as being over local area network, the distributed application programs have been run on two-processor SPARC 20 workstations communicating over a 100 Mbps LAN within the Department of Computing Science at the University of Glasgow. Communications over wide area network took place between a two-processor SPARC 20 workstation in Glasgow and a SPARC Ultra workstation at the Australian National University computing science department in Canberra, Australia.

The total cost of execution of a client program is measured, from the point of invoking the client program's `main` method, until control is returned to the invoker. Thus, the client is initiated from within an already-running VM. In each case, the server is running before the client is invoked and it is shut down after the client terminates. This setup enables a comparison of communication costs over the duration of client program execution, whether all communication of an object graph from the server takes place in one remote call or whether communication of a graph of objects from server to client is done incrementally through the course of client execution. The results of measuring the duration of client execution in each case have been averaged over ten runs, unless stated otherwise. The measurements have been taken in milliseconds but are presented in seconds for readability.

The platform used for the measurements is PJama release version 0.5.7.13 with modifications for object-copying policy support. This version of PJama is a first generation implementation of Orthogonal Persistence for Java, based on JDK 1.1.7.

To illustrate the base costs of persistence and of the object-copying policy support infrastructure, the following graph illustrates the relative costs of running a simple application over a number of platforms. The measured client application looks up a simple `MessageService` at a remote site over a local area network. It makes one remote method call to that service, which passes one ten-character `String` to the server. The RMI call deserialises the `String` at the server and then immediately returns, allowing the client program to complete. Figure 10.1 illustrates the platform costs in the form of a graph. Measurements are given in seconds. Each point on the graph illustrates the cost of execution of the client application over a different platform. The key to the indicated platforms is provided in figure 10.2. Use of each of the policies is illustrated as a range of alternative costs over the "copying-pjama" platform that supports the policy infrastructure, running the application in a `DistributedContext`.

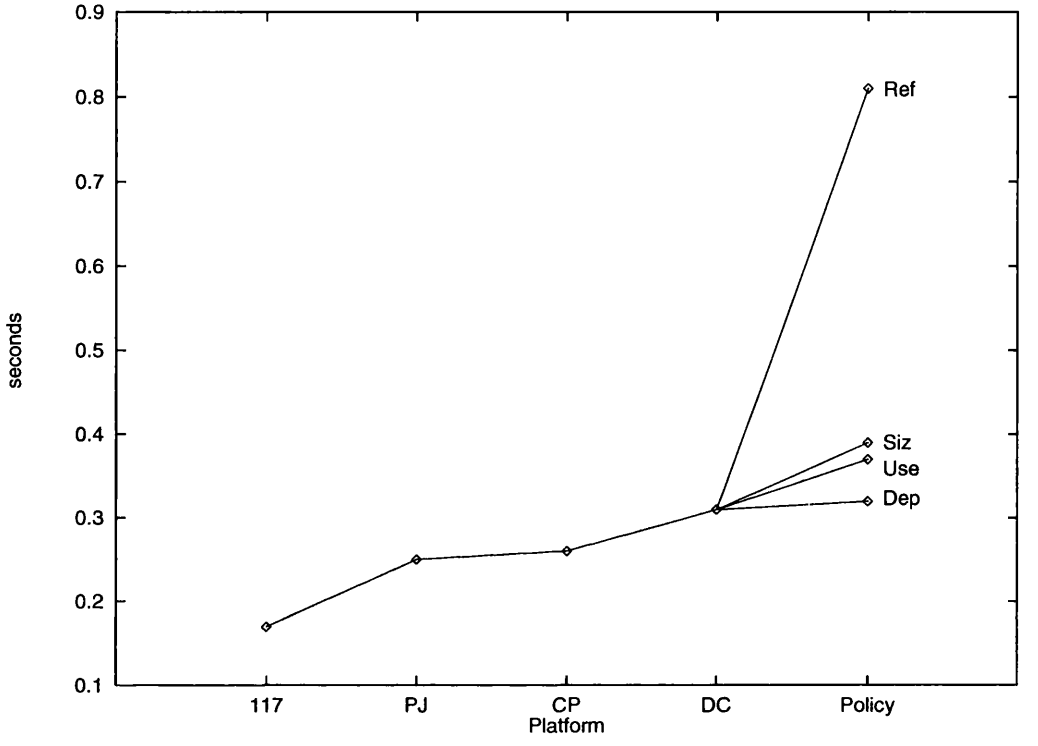

Figure 10.1: Comparison of platform costs

Label	Platform
117	JDK 1.1.7
PJ	PJama 0.5.7.13
CP	PJ + policy support
DC	CP + DistributedContext
Dep	DC + CopyToDepth Policy
Siz	DC + CopyToSize Policy
Use	DC + CopyByUsage Policy
Ref	DC + CopyToRefs Policy

Figure 10.2: Key to platform labels

10.4 How Large is a Large Object Graph?

What constitutes a large object graph, or indeed a large object store, changes as storage and object-oriented database (OODB) technology scales and as application demands become correspondingly more ambitious over time.

The traditional benchmark for OODBs is the 007 benchmark. It was originally used to evaluate several OODBs [CDN93]. The small database contains a module with 500 composite parts, each composite part contains 20 atomic parts and its implementation averages 10MB in size. The medium database contains a module with 500 composite parts, each composite part contains 200 atomic parts and its implementation averages 102MB in size.

Nowadays, the scale of object-oriented databases has increased greatly. In the context of this dissertation, "large" can be interpreted using the following examples. The PJama project aims to support persistent object stores of at least 10GB in size, containing highly structured data. A single graph of all the objects reachable from one root object in a store of that scale could easily be of the order of 30 or 40 MB in size. It is necessary to populate a persistent store incrementally when the volume of data to be stored is too large to create objects for it and make it persistent all in one go. The Geographical Information System developed at the University of Glasgow is a good example of an application that both requires storage of large volumes of data and allows new data to be added to the store incrementally over time. This application stores mapping data. Known as GAP, it was originally developed in Java and subsequently ported to PJama. Stores have been populated incrementally with mapping data from the UK and the US. The project's UK Ordnance Survey data store is about 420MB, while the US TIGER data store for part of California is 1.5GB. The graph of objects reachable from one root in the former contains 699434 objects, totalling 30.45MB in size. Given the availability of US TIGER mapping data, it is possible to add new US states to an existing store as required, during the lifetime of the store. Use of the second generation PJama platform is now increasing the scale of such stores to over 3GB for the TIGER mapping data of the entire state of California and to 4.9GB for an unrelated benchmark called the portable Business Object Benchmark[1].

Given the size of the stores described, the size of the object graphs contained in them also has a tendency to be large. The mapping data of GAP, for example, is composed of lots of small objects that are highly interlinked into large, complex object graphs. Thus, when the copying of object graphs from a persistent store to a remote site is required in a distributed application where such stores are involved, limitations are necessary on the amount of data transferred in one communication between sites. The object-copying policies presented in this paper do support such control: by making copying incremental in the case of the `CopyToRefs`

[1] See section 11.1.6 for more information on the second generation PJama platform

policy, incremental and batched in the case of the `CopyToSize` and `CopyToDepth` policies and based on past usage in the case of the `CopyByUsage` policy. With appropriately chosen parameterised limits, they prevent the objects, of whatever are considered prohibitively-large graphs for the current application, being copied all in one go.

When copying of object graphs between distributed sites is required by an application, the implication is that the costs of doing such copying are outweighed by the benefits to the application of having that copy at its destination. The size and complexity of the GAP application's mapping data certainly argues for caching of local copies rather than repeated remote accesses. Consider the following example. Working with a store of UK Ordnance Survey mapping data, the decision is made to copy an object graph representing a particular map from the server store to the client. The client makes a copy of a map from the server. The map is represented as a root object in the server's store. Copying the full graph of objects reachable from that map root object results in a serialisation of an object graph from server to client that is 1.74MB in size. A measurement has been taken to give an idea of the costs involved in serialisation, communication and deserialisation of such an object graph. A client program contacts the GAP server, makes a deep copy of the full transitive closure of the object graph for the 1.74MB map from the server, using standard RMI serialisation, and then terminates. The time taken for the client to complete is 5.66 seconds.

The original plan was to present the measurements for more controlled copying of this map using the object-copying policies with GAP. The original, single-process application has been converted to a client-server distributed application by the author and a significant amount of work has been done to get GAP working with object-copying policies. However, the complex interactions of this multi-threaded real-world application, with its large graphs of lots of small objects, have proved too much for the current state of the object-copying policy platform. Specifically, handling the interaction between the multi-threading of the GAP client, the Java garbage collector and the remote-faulting support for object-copying policies has proved to be the challenge.

However, the effects of applying the object-copying policies to large object graphs have been successfully measured with several other slightly simpler applications. Experiences with these applications are presented below. Section 10.5 presents two JP applications of the Forest project as an example of two applications that have different requirements for remote usage of the same objects. It describes the object graphs that they use remotely and shows the effects of applying object-copying policies where they seem appropriate. Section 10.6 compares the use of a binary tree of varying sizes over local area network and wide area network, using appropriate object-copying policies.

10.5 Same Object Graph, Different Applications

One of the issues raised by the combination of a static object passing policy and the persistence of objects is that the persistent objects of one class may be used by different applications over time. While the object passing policy originally defined for the class may meet one application's requirements, it may not be as suitable for an application written months or years later to use objects of that same class.

The need for setting an object-copying policy on a per-application basis is demonstrated by a couple of applications that have been developed in the Forest project [For00]. The Forest project aims to provide an environment, known as JP, for the support of large scale software development, which includes distributed configuration management, development and building of applications over sites distributed across wide-area networks [JV97]. Developing applications are managed as federated repositories of versioned software sources. The application software at one site can incorporate specified versions of software available in other repositories; reliable, repeatable builds are supported for the versioned software whether it is all within one repository or distributed across multiple, remote repositories.

The `JPBuild` application enables a user to do a distributed software build. Although the user may have some of their current application's sources under `JP` version control locally, the current version of their application may also use a specific version of software components that are held under JP version control in remote repositories. Thus, in the course of building their application, builds of the required version of each remote software component will be triggered too. Objects in a `JP` repository represent versioned sources. For the purposes of the `JPBuild` application, it is sufficient to always pass objects by reference in the RMI calls that manage the distributed build. To avoid having distribution-related code in JP classes themselves, support was developed in the Forest project for dynamically generating wrappers for the objects to be passed to remote sites. When used, this enforces pass-by-reference semantics at runtime.

However, another application developed for `JP` called the `JPBrowser` benefits more from passing versioned object graphs by a controlled form of copying. The `JPBrowser` supports browsing of local and remote versioned application sources that have been placed under the control of the configuration management system. Use of one of the incremental object-copying policies for browsing remote sources results in the copying of object graphs of names of hierarchies of directories and versioned sources for display to the user by the `JPBrowser`.

For these measurements, `JPBrowser` was initially run to browse over one remote project containing a small set of versioned sources and then subsequently run to browse over a remote project containing a larger set of versioned sources. The serialised size of the full

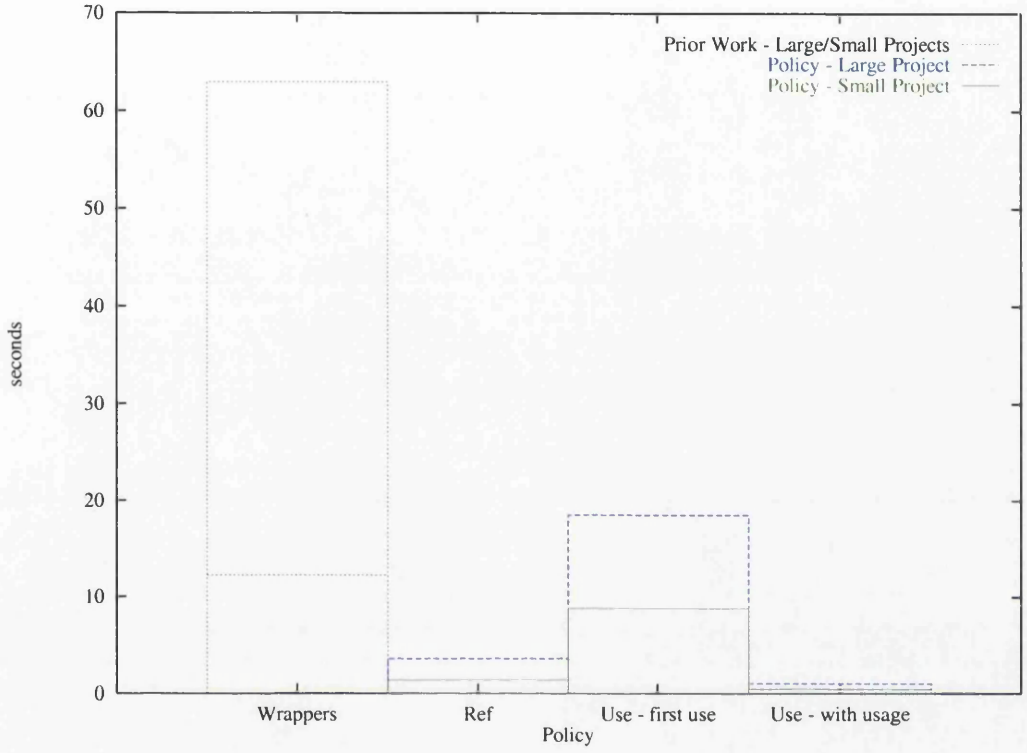

Figure 10.3: Effect of policies on communicating projects

object graph for the small project is 6302 bytes and for the large project is 15.35MB. The measurements have been run between client and server processes, each running on a two-processor SPARC 20 workstation, communicating over a 100 Mbps LAN. The measurements setup is as described in section 10.3.

The results (in seconds) are for the full execution of the `JPBrowser` application. In the first case, the default `JP` wrapping technology developed in prior work of the Forest project is used for *passing by reference* all parameters and results of RMI calls between `JPBrowser` client and the browsed project server. This demonstrates its unsuitability for the `JPBrowser`. The cost of dynamic generation of wrappers to pass objects by reference and the latency cost of every access to the project's objects over the network is high.

In the second case, the `CopyByRefs` object-copying policy is used. The copying of each object on access is less of a penalty than generation of a wrapper for it. Even though there are still latency costs on first client access to each object in the server-side project, the total costs are greatly reduced. For the small project, the cost of using the `CopyToRefs` policy is about a tenth of the cost of the original wrapper technology. For the large project, the `CopyToRefs` policy is about a twentieth of the cost of the original wrapper technology. The effect of this policy on the application is that, as the `JPBrowser` works through the hierarchies of objects representing versioned sources, only their names are accessed and

therefore copied over the network incrementally.

In the third case, the `CopyByUsage` object-copying policy is used, with no existing usage information held at the server for the objects in the project. Note that this policy effectively applies the `CopyToRefs` policy when it has no existing usage information to go on, but that it is also collecting usage information during this first execution using this policy. Thus, the cost of using this policy for the application is six times the cost of `CopyToRefs` for the small project and five times that cost for the large project.

However, in the final case illustrating subsequent executions of the same application with the same `CopyByUsage` object-copying policy, the policy is able to take full advantage of the existing usage information held at the server, which was collected from previous runs on the classes of the project's objects. Here, the objects in the object graph, which are of classes that have been accessed by the client in previous runs, are copied over in one go. The costs, compared with the `CopyToRefs` policy are reduced because less calls are made over the network. In this case, the `CopyByUsage` policy working with previously-collected usage information takes about a third of the time compared to `CopyToRefs` for the application execution using the small project and nearly a quarter of the time for the application execution using the large project. The effect for the `JPBrowser` is that it receives the hierarchy of versioned source names, without the rest of the fields that are associated with the sources, and all in one go. If the user is willing to pay the cost of the first run to gather usage information on a per-class basis, there is obviously some benefit to be had in subsequent use of the same application.

10.6 Same Object Graph, Different Distributed Environments

Another problem, raised by the combination of a static object-passing policy and the persistence of objects, is that one client may access an object in a server-side persistent store over a LAN, while another client may access the same object over a WAN. If the accessed object's class is written with only LAN-scale access envisaged, the manner in which it is passed to remote sites as a parameter in RMI calls may not be as suitable if the persistent object is subsequently used at the scale of a WAN.

Use of an incremental copying policy is attractive when the user does not wish to pay the cost of copying the whole of a large object graph in one go, or does not know how much of the graph will actually be used. However, the benefit of avoiding one large graph copy is offset by the increased latency costs of multiple calls for incremental copying. Running the application over a wide area network, the cumulative latency costs are likely to be high for remote access to the same object graph compared to the same application execution over a

Tree Depth	Nbr Nodes	Serialised Size (bytes)
5	31	767
10	1023	19615
15	32767	622751

Figure 10.4: Size of binary trees at range of depths

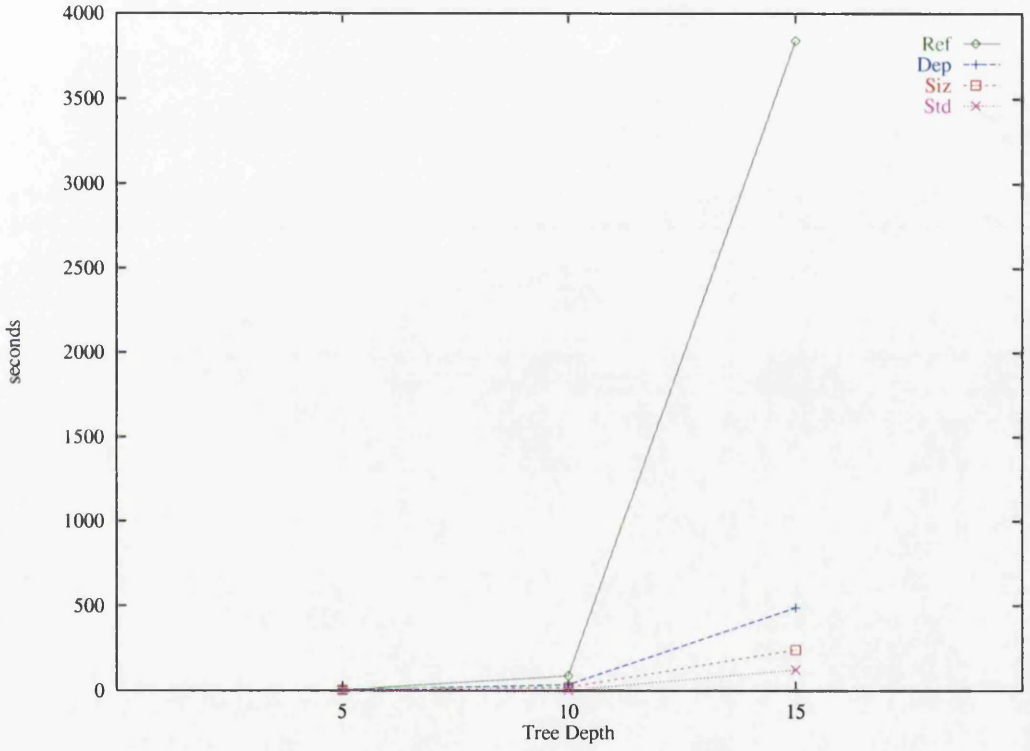

Figure 10.5: Policy-controlled copying over local area network

local area network.

To give the reader some idea about the relative tradeoffs, measurements have been taken on the performance of a client application iterating through a server's binary tree, over both local and wide area networks. For this evaluation, the different client program executions have iterated over binary trees of increasing size. Each object in the tree is an object of 20 bytes in size. It contains three integer fields plus two fields containing references to other nodes in the tree. The table in figure 10.4 indicates, for each depth of tree used, the number of nodes it contains and its serialised size in bytes.

The graphs in figures 10.5 and 10.6 show the cost of client application executions using various object-copying policies to control copying of the object graph of a binary tree of varying sizes, over local and wide area networks respectively.

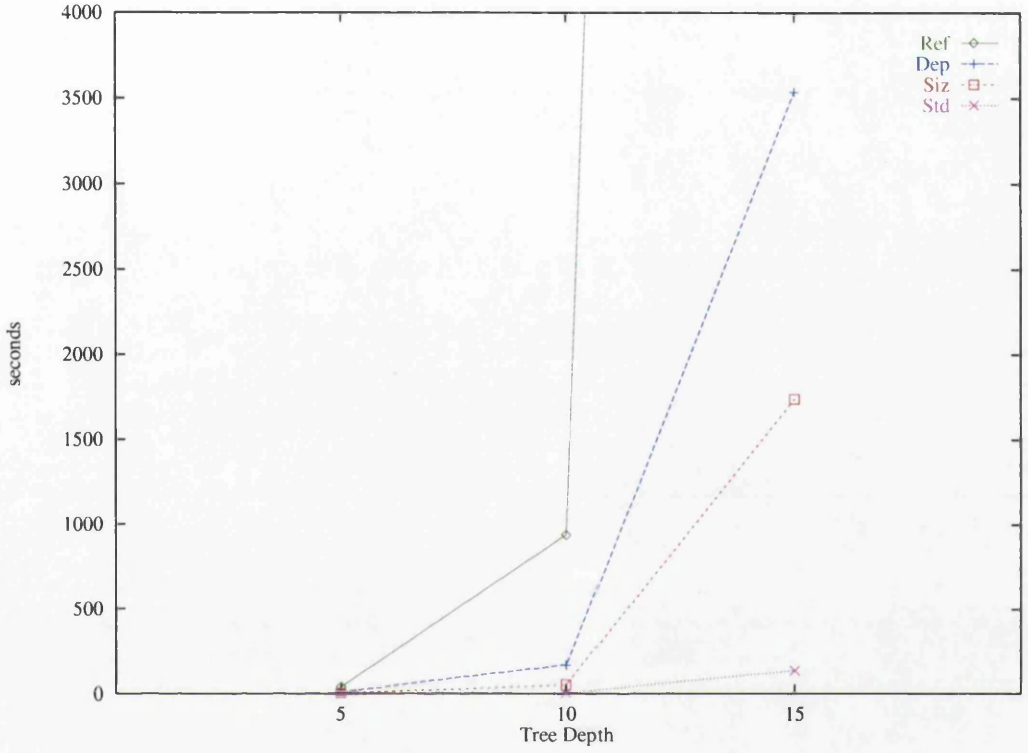

Figure 10.6: Policy-controlled copying over wide area network

The theory is that running the same application in a different distributed environment from the one for which it was originally envisaged may challenge the assumptions made about how objects of a given class should be copied between the participating processes of the application. In the case of this binary tree application, whilst frequent, incremental copying may be fine between processes communicating over a LAN, a more batched mode of copying may be more practical to counter some of the increased latency cost over a WAN, when communicating the same data.

In practice, it is clear that latency costs are certainly a significant factor when considering the performance of the same application over local and over wide area networks. The batched copying modes of the `CopyToSize` and `CopyToDepth` policies are more costly but not necessarily prohibitively so as the latency of the network rises. However, it is clear that the incremental copying mode of the `CopyToRefs` policy is unacceptably costly when latency is high. The graph in figure 10.6 includes an *estimate* of the cost of running the binary tree client program using the `CopyToRefs` policy as it iterates over a binary tree at the server with a depth of 15: it is running off the top of the graph. In fact, the author drew the line at taking any concrete measurement of the cost after the program had already been running for four hours and was still actively copying over the tree incrementally. Extrapolating from the other measurements taken, it is possible that this particular measurement could take up

to ten hours in total if it was allowed to run to completion and didn't crash the machine on which it ran in the course of doing so.

10.7 The Pros and Cons of Object Copying Policies

The advantage of the `CopyToRefs` policy is that it avoids copying over any objects other than those which are accessed by the application. Its disadvantages are the increased network traffic of the many remote calls necessary to do an incremental copy of a graph and the consequent latency costs accrued throughout the time taken to access the graph. This makes it suitable for applications distributed over fast (low latency) networks with a high degree of reliability. The performance penalty of using it across widely-distributed sites with high latency is costly. Comparison of its performance for the `JPBrowser` and for the local area network runs of the binary tree program suggest that this policy works best for incremental copying of a *partial* object graph. There is no advantage to using this policy when the programmer is aware that the whole of the accessed object graphs are likely to be copied over eventually anyway.

The `CopyToSize` policy avoids copying over a greater volume of objects than the serialisation code and destination context can cope with. The disadvantage is the time and space costs of copying over objects which may never be accessed. The batched manner of communication for object graphs performs much better than the `CopyToRefs` policy so it is more suitable for applications running across widely-distributed sites where latency is greater.

The advantage of the `CopyByUsage` policy is the amortisation of the initial cost of collecting usage information over several executions of the same application. Extra time and space costs are incurred though, in recording the usage information. The collected usage information is of benefit if access paths are similar over multiple executions of the same application. This is demonstrated by the measurements taken on the `JPBrowser`, which always accesses the same fields of `JP` versioned source classes. If the object graph is updated radically, this does make usage information redundant for parts of the object graph which become unreachable. Future work on this policy is intended to include refinements to address some of these issues.

It is worth noting that when an object graph is copied incrementally, it is possible for the non-copied parts of the object graph to be modified after the initial top-level copy is made and before the rest of the object graph is accessed. This means that the graph may not be in the same state by the time an individual object is accessed as it was when the the initial copy took place. In fact, the `PCopyStubs` held by clients can reference objects that are no longer reachable from the original graph by the time the `PCopyStub` is accessed. This can

change the semantics of the application.

The intention with evaluation of these policies is that performance is not the sole judge of their value, although this is important. There is plenty of scope for optimisations to the existing policy implementations which could help to bring down their current costs. The intended value to the application programmer is also in the policies' flexible control over how much of an object graph is copied, whether in a single call or over the lifetime of the application. After all, the programmer will not want their client program to have to wait for over five minutes for an object graph of 1.7MB to be serialised with standard RMI when they only want to access selected fields of that object graph. As demonstrated with `JP`, there is some benefit to be gained from use of a well-informed, controlled-copying policy.

10.8 Future Work

10.8.1 New Policies

In the future, policies could be implemented to refine the existing policies and to experiment with new ways of controlling copying between distributed stores.

Further development and optimisation of existing policies is required to explore their potential and costs. One option is to combine the `CopyByUsage` and `CopyToSize` policies on the basis that this is likely to be more useful than either of the existing policies alone. It would ensure that, even if large object graphs have been used in previous application lifetimes, they are not copied over all at once.

There is plenty of scope for the refinement of the `CopyByUsage` policy implementation. Although its implementation currently involves incrementing a `FieldEntry`'s counter on every remote access, it is probably sufficient to stop incrementing once it has reached a value that indicates the field is accessed often. A more subtle use of `FieldEntry.count` would be to serialise the fields of the object corresponding to that `FieldEntry` only if the count is above a certain threshold, where the threshold could be set for a specific class or for the current DistributedContext.

The `CopyByUsage` policy may also benefit from some analysis of object graphs and how they evolve during execution of code working over them. Identifying which parts of the graph always need to be copied and which typically require indicators from the application to determine further accesses would help to minimise unnecessary copying. Notions of articulation points and ownership of (sub)graphs, as described in [PNC98] may also be helpful. However, increasingly sophisticated usage tracking and analysis would have to be balanced against the resultant increasing costs in terms of time and space.

New object-copying policies can be defined and used in the framework of an instance of the class `DistributedContext`, using the hooks described in section 9.4. These could include, for example, a policy which allows an application programmer to specify the application classes for objects that should or should not be copied, during configuration for a `DistributedContext`. Another alternative would be to take an approach based on the programming model of the language Obliq [BC96]. In this case, immutable parts of an object graph are copied but references to mutable objects are passed by reference (replaced with network references).

10.8.2 Shared Subgraphs

CORBA Value Type Semantics, as described in section 4.2.6, and Java RMI both preserve shared subgraphs across the parameters involved in one remote method call. The shared subgraph maintenance currently supported by Java RMI is compromised by object-copying policies that incrementally copy over a graph, since such incremental copying can span a number of remote method calls. If policies are introduced in the future that, for example, apply to some types but not others, the problem is exacerbated. If the policy partially copies one parameter, while leaving the copying of another parameter in the same call to the default Java Object Serialisation (JOS) implementation, shared subgraph maintenance will only be done on the latter object's graph. The reason for this weakening of subgraph maintenance is the intentional separation of object-copying policy from JOS implementation. A tradeoff would need to be made to deal with this issue.

Shared subgraph maintenance within the parameters of one remote method call only partially addresses the issue anyway. It may in fact be more useful to use the limited scope imposed by a `DistributedContext` to manage shared subgraphs between a limited number of distributed sites for the course of a distributed application lifetime. Such support should be provided to the application programmer with similar flexibility to object-copying policies e.g. as an option, so that they only pay the cost if they really need it. The challenge is to provide an implementation which gives good performance and scalability over the lifetime of the application.

Such an implementation would benefit from the use of more unique, system-wide object identifiers than are currently provided by Java. A fingerprint, generated per serialised object, unique for an object in a store, could meet this requirement. Some work is currently in progress in this area [AJ00]. Even without more sophisticated support for shared subgraphs, generation of a fingerprint as the identity of a server-side object would aid tests of equality on multiple stubs that represent the same original object.

10.8.3 Setting A Policy across Multiple Sites

Currently, correct use of policies in a distributed application depends on the application programmer specifying the same policy to be used at all the sites involved. An administration tool for coordinating the setup of distributed contexts would be useful, to *ensure* the same policy is used across all the sites involved.

10.8.4 Measurements

More measurement and evaluation is required of the effectiveness of various policies with a greater range of distributed applications. Experience with real-world applications should contribute to guidelines and recommendations for making the best use of object-copying policies in the future.

10.8.5 Porting

The implementation and use of `PCopyStubs` by object-copying policies relies on the existence of handles to objects in the VM implementation. While every object is accessed via a handle in the VM upon which development of policies has been done so far, this is not the case for the second generation of PJama releases that are based on the Java Solaris Production Release VM [PJR00]. Thus, a redesign would be required for porting this technology to the latest releases of PJama. given the difficulties experienced by the author when trying to measure the copying of large, complex object graphs in a multi-threaded, real-world application running over the first generation platform, the improvements in platform performance, reliability and store capacity, as described in more detail in section 11, would be good incentives for such a port.

More Future Work

More general comments on future work on distribution support for the PJama platform can be found in section 11.

Chapter 11

Future Work

The solutions presented in this dissertation have been implemented and are provided as a platform for distributed, persistent system development. The author has focussed on two issues on the grounds that a complete, integrated solution for persistence and distribution is outwith the scope of a single PhD. However, the solutions provided can be considered the basis for a well-integrated platform. Future work is intended to improve on the existing solutions and to incorporate valuable work in related areas of distribution support, such as distributed consistency management.

Improvements on the existing solutions have been considered.

Future work on application leases for control of dependencies between stores has been presented in context in section 6.3.5. The issues it covers include extension of application leases, the maintenance of leases in the face of store movement from one host to another and the handling of persistent stub objects after lease expiration.

Future work on object copying policies for flexible control over the copying of object graphs between stores has been presented as part of the evaluation of the technology in section 10.8. The issues it covers include improvement of existing policies and development of new ones, handling of shared subgraphs and porting of support for policies to the new, second generation PJama platform.

This section focusses on future work for PJRMI and distribution support for persistent systems in general.

11.1 PJRMI

The ultimate aim of PJRMI is to support resilience of RMI connections between PJVMs within a `DistributedContext`, while also ensuring autonomy of stores by limiting the duration of an application's lifetime in a `DistributedContext`. Aside from the solutions that have been presented to deal with this, some further improvements to the PJRMI implementation are described below.

11.1.1 Reconnection Retries

Since support for persistent remotely-invokable objects and their clients is intended to support their resilience, support for re-tries on re-establishment of a client-server connection after restart should be added to PJRMI, to ensure tolerance of temporary problems with connections.

11.1.2 Store Movement

Persistent stores are likely to move between host machines during their lifetime. Reasons for this may include upgrades to equipment and changes in personnel or to the department within the organisation where the store is being used. For maintainability, persistent RMI objects need, as much as possible, to be associated with the store containing them, rather than the host machine on which that store currently resides. Thus, it should be possible for PJRMI objects to adapt to the movement of a store from one host to another.

A solution has been presented in section 6.2.4.3 for updating the host information for a remotely-invokable object dynamically in the stubs that reference it. It relies on the existence of a third party store lookup process to supply the new location of a store with a given store identity. However, this only deals with changes in host information.

Since the connection information for a remotely-invokable object includes both host and portnumber, there must be support for updating both of these in PJRMI objects at client and server, when necessary. For most remotely-invokable objects, the portnumber on which they are available will change on every store restart. This is already handled by PJRMI's support for renewal of connection information in stubs, on first use by a client after store restart. However, remotely-invokable objects representing well-known services, such as the RMI Registry, are accessed via fixed portnumbers. When a store is moved from one machine to another, it may be found that the portnumber currently used by a well-known service in that store is already in use on the new machine. Thus, extra support is needed for updating the fixed portnumbers of well-known services, as well as for updating the host information.

A PJRMI administration tool would be useful for updating the portnumber of well-known services in a persistent store. Since these services are typically registered as named root objects, they are not hard to locate in the store providing such services, in order to apply such updates. The tool could also be used to inform a client store, that is known to contain references to these services, about the new host information, leaving the client to apply this information to the affected stubs the next time they are used (in order to avoid maintaining an index of them or having to do a scan of a potentially large store to find them).

11.1.3 Persistence of RMI Registry

Some users of PJRMI have had problems with the persistence of the RMI Registry. In theory, they should have the choice over whether or not the RMI Registry persists in their store. However, in practice, the current PJRMI implementation requires it to persist to support look-up by name of the PJRMI implementation service `PJExported`. The `PJExported` service supports re-exportation of stubs on first use after store restart, in order to automatically update the connection information for the corresponding remotely-invokable object. Supporting look-up of this service by name avoids the necessity of making it a well-known service on a fixed portnumber, the issues of which have just been introduced above. If `PJExported` was a well-known service, every client stub would hold a fixed portnumber for it, meaning that every client stub would have to be updated if that portnumber has to be changed.

However, given that the use of application leases effectively limits the lifetime of a client stub, it may be reasonable to set the `PJExported` service to use a fixed portnumber within a `DistributedContext`. Clients are only allowed to update their stubs within the duration of the lease for the application in which they were obtained anyway. Thus, changing the portnumber of `PJExported` between application executions should not be problematic or have unacceptable overheads.

11.1.4 Removing Remote Access to Persistent Objects

PJRMI automatically makes objects persistent when they are exported for remote use. If this exportation is done in a `DistributedContext`, these objects will be unexported when the lease on the current application's lifetime expires. However, if, for some reason, a user wishes to export an object for remote use outwith the control of a `DistributedContext`, it will persist for the lifetime of the store that contains it, even if remote access to it is subsequently removed using the Java RMI unexportation support introduced in JDK1.2 and even if it is no longer reachable from any other application-level persistent object. Extra support

needs to be added to PJRMI to ensure that, if an object is unexported, it will be removed from the PJRMI tracking tables. It may still persist then if reachable from an application-level persistent object but this would ensure that remotely-invokable objects created for relatively short-term use do not persist for the lifetime of the store.

11.1.5 Evolution of Services

As noted in section 3.4.5, a stub becomes unusable once the interface to the service it references has evolved. Some support would be useful to ensure that stubs can be evolved in line with their service implementation. This would be useful for standard Java RMI but is, of course, particularly important for the long-term maintenance of persistent clients and servers using PJRMI.

For a service's store, support for evolving an RMI service class should also cover evolution of its corresponding stub and skeleton classes, plus evolution of any stub class instances that exist there.

For a client's store, evolution support should apply to the stub class and instances, to ensure their continued use with the corresponding, already-evolved remotely-invokable service.

The work of Misha Dmitriev on evolution support for PJama [Dmi98, DA99] and of Huw Evans on DRASTIC [ED97, ED99] provides a good basis for development of such evolution support for PJRMI.

11.1.6 New, Improved PJama Platform

PJRMI will benefit from progress of the PJama platform. An implementation of orthogonal persistence for Java on a new store architecture called Sphere [PAD+98b, PAD98a] has now been released [PJR00]. Amongst other things, the new PJama platform is being used by members of the project to investigate support for persistent threads, technology that improves on use of `PJActionHandlers` for handling externalities [JA99] and support for transactions [DAV97, Day00].

To give some idea of the improvements from which PJRMI can benefit, some statistics for the latest PJama platform, as presented in [Pri00a], are included here. It should be possible to support stores of up to at least 10GB in size. Stores that have actually been built using the new platform include one for a GIS system that loaded the TIGER/Line data [USC98] for the entire state of California (over 3GB store) and another for the portable Business Object Benchmark (pBOB)[BDF+00] (4.9GB store, 24 warehouses, each with 5 threads). The largest single object graph that the platform has been known to handle so far is 34MB (a single scalar array). Speed improvements, in comparison with PJama releases on the

original store architecture, have also been reported. A University of Glasgow student has reported on such improvements, saying "Roughly speaking it varies between 6 times to 16 times faster" [Jap00]. These are mainly due to the introduction of the JIT in the JDK and the much more advanced memory management of Sphere.

Given the size of stores that can now be supported by PJama, long term maintainability is an increasingly important issue for PJRMI.

11.2 Synthesis of Solutions in a DistributedContext

Application leases have been designed to apply to a distributed application where each process is running in a `DistributedContext`. The object copying policy for a distributed application is set and applied within each process's `DistributedContext`. Further development of a `DistributedContext` should therefore include integration of these solutions. The main implication of this integration is that application leases would be set on the `PCopyStub` objects at the leaves of the copied part of an object graph, since they hold the references back to the remote, non-copied parts of the object graph. Such "leased" `PCopyStub` objects would only be usable until the lease runs out. Since an application lease is intended to last for the duration of a distributed application execution, this means that `PCopyStub` objects created as the leaves of partially-copied object graphs are only valid for the duration of that distributed application execution too.

11.3 Additional Support for Persistence and Distribution

Further development of the uses of a `DistributedContext` could include configuration with more distribution-related information and policies on related issues. Setup of an application's `DistributedContext` across multiple sites could include access checks on the sites to be involved. Policies could be incorporated, integrated with the existing support for object copying, for dealing with issues of checkpointing, replication and consistency.

11.3.1 Consistency

Objects are copied across a distributed system for a number of reasons. Depending on the application, the programmer may be happy to make a copy that retains no association with its original. On the other hand, there may be a requirement to maintain consistency between the copy and its original. Much research has been done elsewhere on maintaining consistency of objects across a distributed system. Distribution support for PJama would benefit from

exploring how existing consistency support could be integrated into the platform.

Thor is an example of an existing system that maintains consistency across distributed objects. More details on this work can be found in section 4.2.5. Other work related to this issue includes Arjuna, which provides support for fault-tolerant, distributed systems, using replication of persistent objects, usually in the context of transactions. A brief summary of Arjuna can be found in section 4.2.9. PJama could benefit from the work done on integration of replication support with transactions [LS99b] and with caching [LS99a].

However, any such distribution support provided for PJama should be integrated with solutions addressing the issues of this dissertation. Support for consistency, for example, should be *limited* to within a `DistributedContext` to avoid compromising the long-term autonomy of the stores across which the consistency is being maintained.

It should be noted that support for replication and consistency is likely to require better support for unique identities for objects across distributed VMs than is currently provided in Java.

11.3.2 Transactions

Currently, only one `DistributedContext` runs one application process in a VM at any one time. Given that support is currently being developed for transactions for the PJama platform [DAV97, Day00], the model of a `DistributedContext` will need to be revised in the future to come up with a well-integrated model of usage in a transactional system.

Applying the solutions of this dissertation in a transactional context does require a change in assumptions. It is probably most suitable in the future to apply leases at the level of a store, rather than an individual application, if multiple concurrent applications may run as separate transactions over one store. However, transactions participating in different applications over the same store may wish to use different object copying policies concurrently.

11.3.3 Group Communication

Encompassing individual distributed applications within `DistributedContexts` is likely to result in concurrent groups of cooperative, distributed processes. There is scope for applying the extensive research work that has been done elsewhere on process groups and group communication, as typified by the work of Birman et al. on the Isis and Horus projects [vRBM96].

11.3.4 Aspect-Oriented Programming

The quest for a clean separation between application code and the policies for copying objects between distributed sites can be seen as part of a more general aim to separate out and modularise different concerns within large, complex software systems. The proponents of aspect-oriented programming are well-known amongst those currently pursuing this holy grail [KLM+97]. Aspect programming is supported by AspectJ, which is an aspect-oriented extension to Java [LK99]. PJama's distribution support might be greatly enhanced if it is possible to apply the AspectJ approach to handling of RMI aspects. Identifying the "cross-cutting concern" of, for example, distributed exception handling and implementing it separately from application code could help to free persistent objects from being tied to one specific application context.

11.4 The Big Picture

Ultimately, the challenge in producing a well-integrated persistent, distributed system is to make such a system truly maintainable. The problems raised in this dissertation and by others working in this area are problems that affect the maintainability of persistent stores used in a distributed system. For example, a persistent, distributed system should not, as has been experienced in the past, seize up because the accumulation of dependencies between stores becomes too great and uses up resources unnecessarily. The outstanding questions in this area are:

- what makes a persistent, distributed system maintainable, and
- how can the maintainability of such a system be verified?

Only long-term experience with large persistent stores containing the complex object graphs of real-world applications can confirm whether the challenge has really been met.

Chapter 12

Conclusion

Persistence support has been successfully integrated with distribution support for objects, with greater flexibility that other systems for dealing with two important issues in this area.

A solution has been implemented to address the problems raised by maintaining persistent references between distributed stores. Greater autonomy of individual stores is achieved, by limiting remote access to object graphs to a duration of time associated with a specific distributed application's lifetime. Within the application's lifetime, the benefits are retained of persistence of inter-store references for resilience.

A solution has been implemented to address the problems raised by remote copying of large object graphs. Flexibility of control over such copying is achieved. Separation of object-copying policy from object definition ensures flexibility. Choice of object-copying policy for a specific distributed application's lifetime provides control, while ensuring it is adaptable to changes in size of a persistent object graph over its lifetime and to changes in the context in which that object graph is used.

These solutions address issues that are relevant to the current market place for distributed systems. Global business organisations and E-commerce demand increasingly ambitious distributed software applications with sophisticated data management requirements. Only a platform with well-integrated persistence and distribution support can deliver fast development of such software plus high reliability and maintainability of the result. The importance of such integration is borne out by the coverage of persistence in current industry-standard distributed systems specifications. Strong demand forced early inclusion of mandatory support for persistence in the Enterprise JavaBeans specification [EJB99a]. Demand for a workable specification for persistence for CORBA resulted in the recent adoption of the Persistent State Service specification [OMG99b].

More details on this dissertation's solutions are presented below. Section 12.1 deals with

the implications of creating and maintaining dependencies between distributed, persistent stores. Section 12.2 presents the object-copying policies used to address the problem of large object graph copying between distributed sites.

12.1 Limiting Dependencies Between Stores

The development and use of Persistent RMI (PJRMI), described in chapter 3, has demonstrated that it is feasible to provide the illusion of a persistent connection between two stores. Chapter 6 has explained why it is not possible to *maintain* this illusion for the lifetime of the distributed objects involved.

The PJama platform is intended for use in an open, persistent system. This conforms to the current trend for open, distributed systems that is evident in current use of CORBA in general and Java in particular. CORBA and Java are rapidly becoming the acceptable ways to integrate legacy systems, such as relational databases, into a business's distributed system. A persistent system must be designed to work within this framework, to have any hope of acceptance in the real world. The real world of distributed systems needs to acknowledge that location-transparent use of objects throughout their lifetime is a holy grail, where long-lived objects are concerned. Use of objects in a location-transparent and lifetime-transparent manner makes the programming model simpler but leaves the application programmer with no way to deal flexibly with the distribution-related problems that exist for persistent objects.

Applying persistence by reachability across such an open, distributed system is difficult when not all of the sites involved in an application have support for persistence themselves. Distribution-related errors can prevent successful access to an object, even if it is persistent. The maintainability of a store is dependent on the degree of autonomy it has from other stores; this is compromised by the dependencies this store has with other stores.

The solutions proposed in chapter 6 for dealing with the creation and maintenance of dependencies between stores address both the short-term concerns of the current distributed application's lifetime and the long-term concerns of store maintainability through increased autonomy.

In the short-term, a persistent connection can be maintained between client and server. An application-level lease is set in a wrapper class for the current application's lifetime. The server honours its obligation to provide a remotely-accessible service for the duration of its application lease. The client can determine from stub information that the service runs in a persistence-enabled VM, so that it can afford to make its reference to the service persistent for reliability. It can also determine from stub information when the service is leased for a specified duration, so it knows that it cannot depend on access to the service indefinitely.

In the long-term, the server can remove remote access to a service after its application lease has expired, so that it regains complete control over how and whether to maintain the object itself. The client can determine that the lease on the service it references has run out. This allows it to diagnose service access failures with greater confidence. Withdrawal of service can be distinguished from distribution-related errors. A client can devise a strategy for dealing with withdrawal of service.

The solutions presented here address the problem of unrealistic obligations being placed on stores by support for referential integrity for the lifetime of persistent references to remote objects. PJRMI supports persistent inter-store references within an application's lifetime, while application leases limit remote access to a store's objects, to increase store autonomy with the aim of greater long-term maintainability.

12.2 Policies for Flexible Object Graph Copying Between Stores

The capacity of computers to handle large amounts of data is constantly increasing. Object graphs of at least megabytes in size can now easily be built in main memory but, for reliability and scalability, persistence of object graphs on stable storage is important. These persistent object graphs can be megabytes or even gigabytes in size. PJama now supports object stores of gigabytes in size.

Application programmers may wish to make copies of object graphs for a number of reasons. For example, in a distributed system, particularly one where server load can be high or the latency of network communication is significant, making a copy of a server-side object graph is important to increase availability, reliability and performance for a client.

The issue of object graph copying has been addressed here in the context of copying parameters passed in RMI calls. Where an application programmer requires an object to be passed by copy in an RMI call, they may be aware of the implications of doing so initially. However, if a parameter object is persistent then, over its lifetime, the number of objects reachable from it may grow incrementally. The cost of copying the object graph across the network grows correspondingly.

Lack of flexibility in specification of a remote object access policy has been identified as a problem for persistent objects. It prevents adaptability of this policy over the lifetime of the object to which it is applied, particularly in the face of incremental growth of the graph of objects reachable from it.

Chapter 8 addresses the lack of policy flexibility with support for specifying an object copying policy separately from the definition of the object classes to which it applies. It addresses the handling of passing persistent objects by copy, by enabling a programmer to apply an

appropriate policy to a specific distributed application's lifetime. The programmer chooses, from a selection of object-copying policies, the control over object-copying that is required for the current application. The chosen policy is set in a wrapper class for the current application's lifetime, in a similar manner to the configuration for setting application leases.

As demonstrated in chapter 10, a separation is achieved between application object definition and object-copying policy. The object-copying policies have successfully been applied to control the copying of large object graphs across the network, limiting them by object graph size, object graph depth or past usage. It has also been demonstrated that the same object graphs can be used by different applications and in different distributed environments with object-copying policies appropriate to their context.

The limitations of using these policies for object-graph copying stem mainly from the fact that they are incremental: by its very nature, incremental copying can result in increased network traffic and the possibility of differing application semantics in the face of updates to a graph during its copying to a remote site.

The solutions presented here do address the problems of copying object graphs between stores, when the object graphs may be very large and it may be unnecessary to copy them completely. A number of object copying policies have been implemented that provide control over the copying in different ways. Flexibility has been gained from defining the policies separately from the objects to which they apply. This flexibility does enable the copying of a persistent object graph to be adapted to changes in size and context over its lifetime.

12.3 And Finally...

This dissertation addresses two important issues within the field of distribution support for persistent objects. Realistic solutions have been achieved, which address the problems of trying to maintain long term store autonomy and coping with the remote copying of large object graphs. These solutions require tradeoffs, including the following: application leases limit the persistence of connections between stores in order to increase their long-term autonomy; and policies for copying are incremental to cope with the size of large, persistent object graphs at the expense of performance and, sometimes, differing application semantics. Nevertheless, these solutions make a significant contribution towards the production of well-integrated support for persistence and distribution.

The next challenge is to build an integrated platform based on these existing solutions. The wrapper class, developed for setting an application lease and an object-copying policy on the current application's lifetime, provides a context for plugging in further distribution support in the future. This support could include replication, consistency and checkpointing, for

example. Limiting such support to within the distributed context for a particular application and integrating it with respect for the existing solutions should avoid a recurrence of the problems raised in this dissertation.

This platform should be of great interest to the existing PJRMI user community and to the current business marketplace which has a need for well-integrated, realistic solutions for persistence and distribution.

Appendix A

PJRMI Tutorial

This section contains the documentation for PJRMI at PJama version 0.5.20.2

A.1 Introduction

The first step in implementing support for distribution in PJama is the porting of RMI to the persistent context. A first implementation of persistent RMI (PJRMI) has been produced and is described in this document.

The current implementation of Persistent RMI supports

- the running of standard RMI programs plus
- the running of persistent RMI programs.

  These include support for:

  - persistence of all remotely-invokable objects,
  - lookup by name of remotely-invokable objects that are bound to a name in the Registry,
  - automatic re-exportation of persistent, remotely-invokable objects on first use and
  - automatic reestablishment of the connection between remote, persistent references and remotely-invokable objects on first use of the reference after store restart.

Section A.2 introduces a non-persistent RMI program. Section A.3 then builds on this example to illustrate what changes are necessary to a standard RMI program to make it work in the context of a persistent system.

The following documentation is written in terms of server and client, where the server is the provider of the persistent, remotely-invokable object (service) and the client is the remote user, obtaining and holding a reference to the remote service and making method calls on the service which are remote method invocations.

Other useful information included in this documentation consists of

- section A.4 providing an example of a program that can be used to cleanly shut down a persistent store containing remotely-invokable objects and
- section A.5 containing a list of common exceptions that may be raised during the execution of the example programs in this documentation, each with an explanation of why the exception is likely to have been raised.

Note that the sources for the example code as used in this document are available as part of the PJama release in the directory $PJAMAHOME/demo/pjrmi. Instructions for compiling and running code are given relative to this directory.

A.2 A non-persistent RMI program

The diagram in figure A.1 illustrates, using the example classes introduced below, the objects involved in an RMI call.

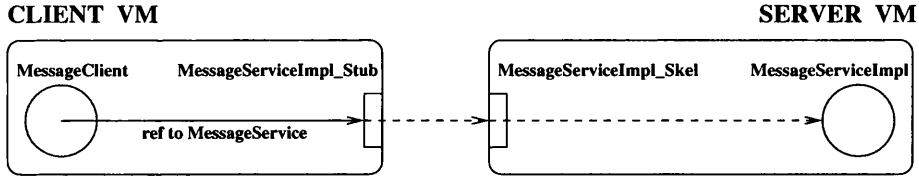

Figure A.1: Objects used for RMI

A.2.1 An RMI-based MessageService

The example used in this document to illustrate the use of RMI uses a remotely-invokable object providing a `MessageService`. This service stores a message as a `String` and provides two remotely-invokable methods: `setMessage` to set the message to a given string and `getMessage` to retrieve the current message. The code in figure A.2 defines a Java interface for this service, suitable for remote use. The code in figure A.3 defines a Java class that implements this interface.

RMI places certain requirements on the definition of the class and interface providing the implementation of a remotely-invokable object.

- A remotely-invokable object can only be accessed remotely via an interface.
- The interface for the remotely-invokable object must implement the `java.rmi.Remote` interface. The RMI implementation relies on the use of this interface `Remote` to determine whether to pass an object by copy or by reference: passing a remotely-invokable object by reference results in the creation of a stub/proxy object in the remote VM.
- The class of the remotely-invokable object must implicitly or explicitly support the exportation of instances of that class to make them remotely usable. The example in figure A.3 gains this functionality by inheriting it from the class `java.rmi.server.UnicastRemoteObject`.
- A `java.rmi.RemoteException` must be thrown by every method of an interface to a remotely-invokable object. This ensures that distribution-related errors that occur during a remote method invocation can be signalled via the throwing of an appropriate exception.
- Where not inherited, the class of a remotely-invokable object is expected to define appropriate methods for `toString`, `equals` and `clone`.

```
package message.service;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MessageService
  extends Remote
{
  public void setMessage(String s)
    throws RemoteException;

  public String getMessage()
    throws RemoteException;
}
```

Figure A.2: Interface MessageService

```
package message.service;

import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class MessageServiceImpl
  extends UnicastRemoteObject
  implements MessageService
{
  private String message;

  public MessageServiceImpl()
    throws RemoteException
  {
    super(); //exports object for remote use
    message = new String("Hello World");
  }

  public void setMessage(String s)
    throws RemoteException
  {
    message = s;
  }

  public String getMessage()
    throws RemoteException
  {
    return message;
  }
}
```

Figure A.3: Class MessageServiceImpl

```
package message.service.nonpersistent;

import message.service.MessageService;
import message.service.MessageServiceImpl;
import java.rmi.Naming;

public class RunService {

  public static void main(String[] args) {
    try {

      MessageService messageService = new MessageServiceImpl();
      Naming.rebind("MessageService", messageService);
      System.out.println("MessageService ready for remote use");

    } catch (Exception e) {
      System.out.println("RunService.main: exception raised: ");
      e.printStackTrace();
    }
  }
}
```

Figure A.4: class RunService creates MessageService

In order to create an instance of a `MessageService` and make it available to support remote invocations on the methods in its interface, the following steps must be taken:

1. Compile service files
   `javac message/service/MessageService.java`
   `message/service/MessageServiceImpl.java`
2. Generate RMI files
   `rmic message.service.MessageServiceImpl`
   produces `message/service/MessageServiceImpl_Stub.class`,
   `message/service/MessageServiceImpl_Skel.class`
3. Run name service
   `rmiregistry &`
4. Run a program to make a `MessageService` available for remote use.

An example program that creates and registers the service is shown in figure A.4. It can be compiled and run using the following commands:

```
javac message/service/nonpersistent/RunService.java
java message.service.nonpersistent.RunService
```

This program has two significant steps:

(a) Create `MessageService` (exports it for remote use)
```
MessageService messageService = new MessageServiceImpl();
```

(b) Register object by name
```
Naming.rebind("MessageService", messageService);
```
This then allows clients to do a look up by name to obtain a reference to the published `messageService`.

The execution of the program `RunService` on the machine called kona should produce the following output:

```
susan@kona: java message.service.nonpersistent.RunService
MessageService ready for remote use
```

A.2.2 A non-persistent client for the MessageService

An object in a different VM from the one where the `MessageService` has been created needs to obtain a reference to the service before it can use it. The code in figure A.5 defines a client that, given a reference to a `MessageService` as an argument to its constructor, supports one method to report the current message held at a `MessageService` and change it to a new one.

An example program that creates and uses the `MessageClient` is shown in figure A.6. In order to create and use the `MessageClient`, the following steps must be taken:

1. Compile client files
   ```
   javac message/client/MessageClient.java
         message/client/nonpersistent/RunClient.java
   ```

2. Run client program, in this example supplying the name of the host where the `MessageService` should be available for remote use and the new message to set at the `MessageService`.
   ```
   java message.client.nonpersistent.RunClient kona.dcs.gla.ac.uk
   ''This is a new message''
   ```

```
package message.client;

import message.service.MessageService;

public class MessageClient
{
  private MessageService msRef;

  public MessageClient(MessageService ms) {
    try {
      msRef = ms;
    } catch (Exception e) {
        System.out.println("MessageClient: exception occurred:");
        e.printStackTrace();
    }
  }

  public void changeMessage(String newMessage) {
    try {
      String oldMessage = msRef.getMessage();
      msRef.setMessage(newMessage);
      String checkedMessage = msRef.getMessage();
      System.out.println("MessageClient: message changed from"
                         + oldMessage + "to " + checkedMessage);
    } catch (Exception e) {
      System.out.println("MessageClient: exception occurred:");
      e.printStackTrace();
    }
  }
}
```

Figure A.5: MessageClient uses MessageService

```
package message.client.nonpersistent;

import message.service.MessageService;
import message.client.MessageClient;
import java.rmi.Naming;

public class RunClient{

  public static void main(String[] args) {
     try {

      String service = new String("//");
      try {
        service = service.concat(args[0]);
      } catch (ArrayIndexOutOfBoundsException ae) {
        System.out.println("\nUsage: RunClient <servername> <message>");
        System.exit(-1);
      }
      service = service.concat("/MessageService");
      System.out.println("RunClient: using service " + service);

      MessageService msRef = (MessageService) Naming.lookup(service);
      MessageClient messageClient= new MessageClient(msRef);
      messageClient.changeMessage(args[1]);

    } catch (Exception e) {
      System.out.println("RunClient.main: exception raised: ");
      e.printStackTrace();
    }
  }
}
```

Figure A.6: RunClient creates and uses MessageClient

This program has three significant steps:

(a) Lookup service by name

```
MessageService msRef = (MessageService) Naming.lookup(service);
```

(b) Create client to use service

```
MessageClient messageClient= new MessageClient(msRef);
```

(c) Use service

```
messageClient.changeMessage(args[1]);
```

The execution of the program `RunService` should produce the following output:

```
susan@hawaii: java message.client.nonpersistent.RunClient
kona.dcs.gla.ac.uk ''This is a new message''
RunClient: using service //kona.dcs.gla.ac.uk/MessageService
MessageClient: message changed from ''Hello World'' to
''This is a new message''
```

The standard RMI interface `Naming` provides a method `lookup` which, given a URL supplying the name of the service and the name of the host where the service is located, will obtain and return a stub object representing that service for use by the client. Note that the class of the service from the client's point of view is that of the *interface* to the service.

A.3 A persistent RMI program

A.3.1 Creating and using persistent, remotely-invokable objects

The previous section introduced an example of a program that uses standard RMI to create an object that supports the `MessageService` interface and make it available for remote use. The modifications necessary to provide `MessageService` as a *persistent*, remotely-invokable object are now described.

Firstly, a small change in programming model is recommended. In the standard RMI program, the `MessageService` was created, which also automatically exports it for remote use, and then the program runs indefinitely, waiting to service incoming method calls from other VMs. If the program execution is killed, then the next time the `MessageService` is required, the program must be run again, creating and exporting the `MessageService` again, and again it runs indefinitely. In the persistent RMI model, we would like to create the service once, make it persistent and then have it available in the persistent store to service

appropriate incoming method calls during future sessions that use that store. Thus, we recommend two distinct stages in persistent RMI: in the first stage a service is created and made persistent; in the second, an existing persistent service is available for remote use. To model these stages, we have two programs: the first program runs over a persistent store, creates a `MessageService`, exports it for remote use and makes it persistent; the second runs over the same store, making the persistent services in that store available for remote use.[1]

A.3.1.1 Populating the persistent store with support services

Before presenting the code for creating and using a persistent application service, here are the details for setting up a persistent store and populating it with a couple of remotely-invokable objects that will be of general use to programs using persistent RMI. The store is set up using the following steps:

1. Compile the support service classes
   ```
   javac support/service/persistent/SuspendService.java
         support/service/persistent/SuspendServiceImpl.java
         support/service/persistent/CreateSupportServices.java
   ```

2. Generate RMI files
   ```
   rmic support.service.persistent.SuspendServiceImpl
   ```

3. Create a persistent store using the appropriate tool `opjcs` provided as part of the PJama release (Note: change the path and storename to ones which are appropriate to your environment)
   ```
   opjcs /local/stores/services.pjs
   ```
   This creates a store which is written to disk in the file /local/stores/services.pjs. By convention storenames are postfixed with `.pjs`.

   Note: to *recreate* an existing store there is an `overwrite` option to `opjcs`:
   ```
   opjcs /local/stores/services.pjs -overwrite
   ```

4. Run the program `CreateSupportServices`, illustrated in figure A.7, to create a couple of standard persistent RMI services:
   ```
   opj -Xstore /local/stores/services.pjs
   support.service.persistent.CreateSupportServices
   ```

[1] The alternative to the recommended model is to create and populate a store with remotely-invokable objects and make them available for remote use all in one program. However, it is useful to be able to separate population of a store from use of the objects in the store since normal usage often involves populating a store once and then using the store contents repeatedly.

```
package support.service.persistent;

import java.rmi.registry.Registry;
import sun.rmi.registry.RegistryImpl;
import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;
import org.opj.store.PJActionHandler;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Naming;

public class CreateSupportServices{

  public static void main(String[] args) {
    int portnumber;

    try {
      if (args.length > 0) {  //user specified Registry port number
        portnumber = Integer.parseInt(args[0]);
      }
      else {
        portnumber = Registry.REGISTRY_PORT;
      }

      Registry registry = new RegistryImpl(portnumber);
      PJStore pjs = PJStoreImpl.getStore();
      pjs.newPRoot("Registry", registry);
      (PJStoreImpl.getActionManager()).bind((PJActionHandler)registry);

      SuspendService suspendService = new SuspendServiceImpl();
      UnicastRemoteObject.exportObject(suspendService);
      Naming.rebind("SuspendService", suspendService);

      System.exit(0);
    } catch (Exception e) {
      System.out.println("CreateSupportServices.main: exception raised: ");
      e.printStackTrace();
    }
  }
}
```

(a) A `Registry` is created, which includes automatically exporting it for remote use, registered as a persistent root and registered with the `PJActionManager`. (See PJama API documentation for information for information about the `PJActionManager` and the need for `PJActionHandlers` associated with some classes.)

(b) A `SuspendService` object is created, exported and bound to a name. This service has just one method `suspendAndQuit` which suspends all currently running threads (including all the threads associated with exported, remotely-invokable objects listening for incoming method calls), stabilises the persistent store and terminates the current execution of the VM.

(c) The call `System.exit(0)` is made explicitly to terminate the potentially-indefinite running of the threads associated with the exported, remotely-invokable objects and stabilise the persistent store, which includes capturing the state of the newly-persistent objects.

The explicit call to the static method `UnicastRemoteObject.exportObject` made for the object `suspendService` demonstrates the alternative way to make objects available for remote invocation. While, in section A.2 the `class MessageServiceImpl extends UnicastRemoteObject` in order to inherit the code necessary for automatically exporting objects of that class on creation of an instance of the class, the static `exportObject` method of `UnicastRemoteObject` supports the exportation of, in principal, *any* object, at any point during the life of that object. The other RMI restrictions, as specified in section A.2, do still apply to the class of that object though.

The persistence of instances of the `Registry` and `SuspendService` simplify the running of programs that use RMI. Creating and making persistent an instance of the `Registry` means that it is no longer necessary to start an `rmiregistry` process running before invoking a program that creates and uses remotely-invokable objects.

Note that support is not now provided for dynamic, automatic generation of the `_Stub` and `_Skel` classes associated with remotely-invokable objects. As with standard RMI, remember to invoke the `rmic` compiler to generate SuspendServiceImpl_Stub.class and SuspendServiceImpl_Skel.class before running the persistent RMI program `CreateSupportServices`. Do the same for MessageServiceImpl_Stub.class and MessageServiceImpl_Skel.class in between compiling the application classes and running the `MessageService` creation program.

A.3.1.2 Populating the store with persistent, remotely-invokable application objects

Now that we have a persistent store prepared to support persistent, remotely-invokable objects, we can run a program to create some. Firstly, it is important to note that the code of the `interface MessageService` and the `class MessageServiceImpl` used by the persistent RMI program is unchanged from that used by the standard, non-persistent RMI program; it is still the code as illustrated in figures A.2 and A.3.

The code in figure A.8 is an example of a persistent RMI program that creates a `MessageService`, exports it for remote use and makes it persistent. This program is run

```
package message.service.persistent;

import message.service.MessageService;
import message.service.MessageServiceImpl;
import java.rmi.Naming;

public class CreateService{

  public static void main(String[] args) {
    try {

      MessageService messageService = new MessageServiceImpl();
      Naming.rebind("MessageService", messageService);
      System.exit(0);

    } catch (Exception e) {
      System.out.println("CreateService.main: exception raised: ");
      e.printStackTrace();
    }
  }
}
```

Figure A.8: CreateService creates persistent MessageService

as follows:

1. Compile the service classes.
   (Only necessary to compile `MessageService*` files if not already compiled for non-persistent program.)

```
javac message/service/MessageService.java
      message/service/MessageServiceImpl.java
      message/service/persistent/CreateService.java
```

2. Generate RMI files.
   (Only necessary to generate MessageServiceImpl_Stub.class and MessageServiceImpl_Skel.class if not already generated for non-persistent program.)
   `rmic message.service.MessageServiceImpl`

3. Run the program to create the service, over our existing persistent store
   `opj -Xstore /local/stores/services.pjs`
   `message.service.persistent.CreateService`

   (a) The creation of the `MessageService` object includes the object's exportation for remote use, since the `class MessageServiceImpl` inherits this functionality from the class `UnicastRemoteObject`. Exportation of the object automatically registers it with a persistent table, thus also making the `MessageService` object persistent too.

   (b) The call to `Naming.rebind` supports remote lookup of the `MessageService` by name.

   (c) The call `System.exit(0)` is made explicitly to terminate the potentially-indefinite running of the threads associated with the exported, remotely-invokable objects and stabilise the persistent store, which includes capturing the state of the newly-persistent objects.

   The program should complete with no output.

A.3.1.3 Using existing persistent, remotely-invokable objects

Having populated the store with persistent, remotely-invokable objects, one program can be used repeatedly to open a session over the store where the persistent, remotely-invokable objects will be automatically available for remote use. The code in figure A.9 is an example of such a program. To use this program, the following steps are taken:

1. Compile program
   `javac message/service/persistent/UseService.java`

```
package message.service.persistent;

import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;

public class UseService {

  public static void main(String[] args) {
    try {

      PJStore pjs = PJStoreImpl.getStore();
      if (pjs != null) {
        // maintain the running of this JVM indefinitely
        // to service incoming method invocations
        while (true) {
          try {
            System.out.println("\nUseService.main:
              waiting for incoming connections");
            //copied from sun.rmi.registry.RegistryImpl
            Thread.sleep(Integer.MAX_VALUE - 1);
          } catch (InterruptedException e) {}
        }
      }
      else
        System.out.println("UseService.main: no store");

    } catch (Exception e) {
      System.out.println("UseService.main: exception raised: ");
      e.printStackTrace();
    }
  }
}
```

Figure A.9: UseService makes persistent, remotely-invokable objects available

2. Run it to make the persistent, remotely-invokable objects available

```
opj -Xstore /local/stores/services.pjs
message.service.persistent.UseService
```

The program should give the following output:

```
susan@kona: opj -Xstore /local/stores/services.pjs
message.service.persistent.UseService
UseService.main: running...
```

Each persistent, remotely-invokable object, with which we populated our example store, is automatically re-exported the first time that a client tries to access it. The program above just ensures that the session running over the store continues running as long as it is required. The `SuspendService` of this store can be used from a different VM to terminate the session of this VM, when the services available from this store are no longer required for the time being. (See the latter part of section A.3.2 for information on using the `SuspendService`.)

The next time an application programmer wants these services to be available again, all they have to do is rerun this `UseService` program and the re-establishment of the services is done automatically once again.

A.3.2 Creating and using persistent references to remote, remotely-invokable objects

Just as remotely-invokable objects can be made persistent, references to them can also be made persistent. The program with the code in figure A.5 is the client for the `MessageService`, that was introduced in the non-persistent program example; as described before, it has one method `changeMessage` that changes the message held at the server's `MessageService` and reports what message is held by the `MessageService` before and after it is changed.

Just as a two-stage model of use was proposed for the creation and use of remotely-invokable objects, the same idea is proposed here for the creation and use of a persistent reference to a remote, remotely-invokable object. The program with the code in figure A.10 creates a `MessageClient` object, as defined in A.5, including the establishment of a reference to a `MessageService`, and makes it persistent. The program is used as follows:

1. Compile the client classes.
   (Only necessary to compile `MessageClient.java` if not already compiled for non-persistent program.)
   ```
   javac message/client/MessageClient.java
         message/client/persistent/CreateClient.java
   ```

2. Create a persistent store for the client program to use
   ```
   opjcs /local/stores/clients.pjs
   ```

3. Run the program to create the `MessageClient` and make it persistent, in this example supplying the name of the host where the `MessageService` should be available for remote use.
   ```
   opj -Xstore /local/stores/clients.pjs
   message.client.persistent.CreateClient kona.dcs.gla.ac.uk
   ```

   (a) A reference to the `MessageService` is obtained from the specified host, using the standard call to `Naming.lookup`.

   (b) A `MessageClient` is created and supplied with the `MessageService` reference.

   (c) The new instance of `MessageClient` is made persistent.

   The program should give the following output:

   ```
   susan@hawaii: opj -Xstore /local/stores/clients.pjs
   message.client.persistent.CreateClient kona.dcs.gla.ac.uk
   Client using service: //kona.dcs.gla.ac.uk/MessageService
   ```

A.3.2.1 Using an existing, persistent reference to a service

To use an existing, persistent reference to a remote, remotely-invokable object, an application program can be written that uses the reference just as if it were a local reference. Any re-establishment of connections between the persistent reference and the remote, remotely-invokable object that is necessary is done automatically *the first time the persistent reference is accessed* after the reopening of the persistent store containing the reference. The example program with the code in figure A.11 demonstrates the use of a persistent `MessageClient` that contains and uses a persistent reference to a `MessageService`. To use the program:

1. Compile the client class
   ```
   javac message/client/persistent/UseClient.java
   ```

```
package message.client.persistent;

import message.service.MessageService;
import message.client.MessageClient;
import java.rmi.Naming;
import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;

public class CreateClient {

  public static void main(String[] args) {
     try {

      String service = new String("//");
      try {
        service = service.concat(args[0]);
      } catch (java.lang.ArrayIndexOutOfBoundsException ae) {
        System.out.println("\nUsage: CreateClient <servername>");
        System.exit(-1);
      }
      service = service.concat("/MessageService");
      System.out.println("Client using service: " + service);

      MessageService msRef = (MessageService) Naming.lookup(service);
      MessageClient messageClient= new MessageClient(msRef);

      PJStore pjs = PJStoreImpl.getStore();
      pjs.newPRoot("MessageClient", messageClient);

    } catch (Exception e) {
      System.out.println("CreateClient.main: exception raised: ");
      e.printStackTrace();
    }
  }
}
```

Figure A.10: CreateClient creates a persistent MessageClient

```
package message.client.persistent;

import message.client.MessageClient;
import org.opj.store.PJStore;
import org.opj.store.PJStoreImpl;

public class UseClient {

  public static void main(String[] args) {
    try {

      PJStore pjs = PJStoreImpl.getStore();
      MessageClient messageClient =
        (MessageClient) (pjs.getPRoot("MessageClient"));
      try {
        messageClient.changeMessage(args[0]);
      } catch (java.lang.ArrayIndexOutOfBoundsException ae) {
        System.out.println("\nUsage: UseClient <messageString>");
        System.exit(-1);
      }
    } catch (Exception e) {
      System.out.println("UseClient.main: exception raised: ");
      e.printStackTrace();
    }
  }
}
```

Figure A.11: UseClient uses MessageClient

2. Run the client program, supplying a String to change the message to at the MessageService
   `opj -Xstore /local/stores/clients.pjs message.client.persistent.UseClient ''Working persistent service and client - hurray''`

   (a) The MessageClient is looked up by name in the persistent store.

   (b) The method changeMessage is called, passing the given string as a parameter. It will use the persistent reference to a MessageService to change its message.

   This program will give the following output:

```
susan@hawaii: opj -Xstore /local/stores/clients.pjs
message.client.persistent.UseClient ''Working persistent
service and client - hurray''
MessageClient: message changed from "Hello World" to
"Working persistent service and client - hurray"
```

A.4 Using the SuspendService to close down a persistent store

As in standard RMI, a program running over a persistent store supporting remotelyinvokable objects will run indefinitely, until it is interrupted or killed. However, an alternative to this is to make use of the SuspendService to close down the store cleanly. A clean shutdown ensures that persistent objects or updates to existing persistent objects are really made persistent.

The interface in figure A.12 supports remote invocation of a method to shut down a store cleanly. It may be implemented as in figure A.13. The code in figure A.14 is an example

```
package support.service.persistent;

public interface SuspendService extends java.rmi.Remote
{
  public void suspendAndQuit()
    throws java.rmi.RemoteException;
}
```

Figure A.12: SuspendService

of a client program that gets a reference to the SuspendService and then calls its method suspendAndQuit to close down the store cleanly. Note that in the implementation of figure A.13 a thread is started to close down the store, separately from the thread executing the

```
package support.service.persistent;

public class SuspendServiceImpl implements SuspendService
{
  public void suspendAndQuit()
    throws java.rmi.RemoteException
  {
    RealSuspendService realSuspendService = new RealSuspendService();
    new Thread(realSuspendService).start();
  }
}

class RealSuspendService implements Runnable
{
  public void run() {
    try {
      System.out.println("RealSuspendService running...");
      Thread.sleep(30000);  // 30 secs
      System.exit(0);
    } catch (Exception e) {
      System.out.println("RealSuspendService.run: exception raised "
        + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Figure A.13: SuspendServiceImpl implements SuspendService

remotely-invoked method `suspendAndQuit`. This allows the remote method execution to complete and return cleanly before the store is closed down.

A.5 RMI Exceptions

If one of the following exceptions is raised, this section may help in a diagnosis of the problem.

A.5.1 java.lang.ClassNotFoundException

The exception `java.lang.ClassNotFoundException` may be raised when a client, having made a remote method call, tries to receive a stub object as the return value but cannot find its corresponding class. Ensure that the stub class is available to the client, e.g. included in its `CLASSPATH`.

A.5.2 java.rmi.server.ExportException

The exception `java.rmi.server.ExportException` may be raised when a server tries to export an object for remote use. Ensure that appropriate stub and skeleton classes have been generated for the object being exported. Remember that it is necessary to call the `rmic` compiler to generate the required class files before running the program that uses them.

A.5.3 java.lang.IllegalAccessException

The exception `java.lang.IllegalAccessException` may be raised if a server picks up the wrong version of stub files generated to support PJRMI operation. Ensure the PJRMI classes rather than standard JDK RMI classes are being used i.e. if both appear in the CLASSPATH, check the PJRMI classes appear first.

A.5.4 java.lang.NullPointerException

The exception `java.lang.NullPointerException` may be raised at a client when it tries to use a persistent reference to a remote, remotely-invokable object. Ensure that the stub held by the client has been generated by PJRMI. Since the stubs representing remotely-invokable objects have slightly different functionality for PJRMI, stubs generated by standard JDK RMI code will not always work after they have been made persistent.

```
package support.client.persistent;

import support.service.persistent.SuspendService;
import java.rmi.Naming;

public class SuspendClient {

  public static void main(String[] args) {
    try {

      String service = new String("//");
      try {
        service = service.concat(args[0]);
      } catch (java.lang.ArrayIndexOutOfBoundsException ae) {
        System.out.println("\nUsage: CreateClient <servername>");
        System.exit(-1);
      }
      service = service.concat("/SuspendService");
      System.out.println("Client using service: " + service);

      SuspendService ssRef = (SuspendService) Naming.lookup(service);
      ssRef.suspendAndQuit();

    } catch (Exception e) {
      System.out.println("SuspendClient.main: exception raised "
        + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Figure A.14: SuspendClient uses SuspendService

A.6 Comments

Your feedback on the current implementation of PJRMI would be appreciated.

- If you find any errors in the documentation, think something in the documentation could be explained better or think something should be added to the documentation or
- if there is something about the design of PJRMI that you think should be changed

please email comments to susan@dcs.gla.ac.uk

Appendix B

PJActionHandler Usage

```
summary
-------
20 classes with PJActionHandlers
Majority registered during static initialisation  (16/20)
All but one called on store restart
6 reinitialise native variables
7 reinitialise static variables
Rest used for other tasks that usually require execution
*on* store restart rather than lazily after it


When registered
---------------
static init:
java/lang/ClassLoader.java
java/net/InetAddress.java
java/net/PlainSocketImpl.java
java/net/SocketInputStream.java
java/net/SocketOutputStream.java
java/net/DatagramPacket.java
java/net/PlainDatagramSocketImpl.java
java/awt/Window.java
java/rmi/dgc/VMID.java
sun/misc/Launcher.java
```

```
sun/rmi/transport/tcp/TCPTransport.java
sun/rmi/transport/tcp/TCPEndpoint.java
sun/rmi/transport/DGCImpl.java
sun/rmi/transport/ObjectTable.java
sun/rmi/transport/DGCClient.java
sun/rmi/transport/DGCAckHandler.java

instance method:
org/opj/store/ObjectCacheObserverStarter
org/opj/hidden/PJSharedTraceFile.java
org/opj/distribution/PJamaPJExported.java

explicitly by app:
sun/rmi/registry/RegistryImpl.java


When called
-----------
on stabilisation:
java/lang/ClassLoader.java

on store restart:
java/net/InetAddress.java
java/net/PlainSocketImpl.java
java/net/SocketInputStream.java
java/net/SocketOutputStream.java
java/net/DatagramPacket.java
java/net/PlainDatagramSocketImpl.java
java/awt/Window.java
java/rmi/dgc/VMID.java
sun/misc/Launcher.java
sun/rmi/registry/RegistryImpl.java
sun/rmi/transport/tcp/TCPTransport.java
sun/rmi/transport/tcp/TCPEndpoint.java
sun/rmi/transport/DGCImpl.java
sun/rmi/transport/ObjectTable.java
sun/rmi/transport/DGCClient.java
sun/rmi/transport/DGCAckHandler.java
```

```
org/opj/store/ObjectCacheObserverStarter
org/opj/hidden/PJSharedTraceFile.java
org/opj/distribution/PJamaPJExported.java


Task
----
reinit native variables:
java/net/InetAddress.java
java/net/PlainSocketImpl.java
java/net/SocketInputStream.java
java/net/SocketOutputStream.java
java/net/DatagramPacket.java
java/net/PlainDatagramSocketImpl.java

reinit static variables:
java/rmi/dgc/VMID.java
sun/rmi/transport/tcp/TCPTransport.java
sun/rmi/transport/tcp/TCPEndpoint.java
sun/rmi/transport/DGCImpl.java
sun/rmi/transport/ObjectTable.java
sun/rmi/transport/DGCClient.java
sun/rmi/transport/DGCAckHandler.java

misc:
java/lang/ClassLoader.java
  mark fields persistence transient
java/awt/Window.java
  recreates every visible Window instance
sun/misc/Launcher.java
  resets classpath and classloader for main thread
sun/rmi/registry/RegistryImpl.java
  manually reexport for remote use
org/opj/store/ObjectCacheObserverStarter
  relaunches object cache observer if to be used
org/opj/hidden/PJSharedTraceFile.java
  reinstalls tracing if to be used
org/opj/distribution/PJamaPJExported.java
```

```
manually reexports for remote use
```

Appendix C

Object Copying Policy Support

C.1 The Lifetime of a PCopyStub

This section provides extra detail, to that given in section 9.3, on the support provided for a `PCopyStub` at different points in its lifetime.

A `PCopyStub` is first created during serialisation of an object graph for an RMI call, where an object-copying policy determines that, instead of copying the rest of the reachable objects from a point in the graph, a `PCopyStub` placeholder should be inserted instead.

C.1.1 Deserialisation of PCopyStub

On deserialisation of this modified object graph at its destination, the `PCopyStub` is initially created as a normal object in memory. At the JVM implementation level, the handle to this `PCopyStub` object is also initially in normal object format. However, in order to ensure that any accesses made to the `PCopyStub` trigger a remote fault of the corresponding original object, the format of the `PCopyStub` handle is modified before deserialisation is completed. The address of the `PCopyStub` object held by the handle is modified, so that access to a `PCopyStub` fails the initial residency check. The address is incremented by one, making it look like a PID rather than a normal, eight-byte-aligned Java object. The methods of the `PCopyStub` class, referenced from the other field of the handle, are moved to the `tmp` field of the `PCopyStub` object itself. The handle's `methods` field can then be set with a typecode set to `T_PROXY`. This leaves the handle to the `PCopyStub` in a remote fault format as illustrated in step one of figure C.1.

C.1.2 Residency check on PCopyStub in GC Heap

When access is attempted to a PCopyStub in remote fault format, the non-eight-byte-aligned object address fails the initial residency check. This indicates that this is not a normal, memory-resident Java object. The typecode set to T_PROXY distinguishes this remote fault format object from a non-resident object in local fault format. A remoteObjectFault function is called to accomplish the remote fault of the object represented by the PCopyStub.

The first stage of the remote faulting code resets the handle of the PCopyStub from remote fault format to normal object format, as illustrated in the first two stages of step two in figure C.1. A call to the method PCopyStub.getRealObject returns the remote object represented by this PCopyStub. To ensure that all objects that referenced the PCopyStub handle locally can now reference the returned remote object instead, the fields of the PCopyStub handle are overwritten to hold the newly-faulted object and the methods of its class directly. This is illustrated in the last stage of step two in figure C.1.

C.1.3 Promotion of a PCopyStub from the GC Heap

If a PCopyStub, still in remote fault format, becomes reachable from a persistent object, it will be promoted in-memory from the Java garbage-collected heap to the persistent object cache, as well as being written to the persistent store on disk. Step three of figure C.1 illustrates the format of the handle to a PCopyStub during this promotion. A PCopyStub is identified during promotion by its typecode set to T_PROXY. A macro is called at this point to reset the PCopyStub's handle back from remote fault format to normal object format. This then allows the PCopyStub itself to be promoted like any other normal object.

Unlike objects in the Java GC Heap, which are referenced via a JHandle, objects in the object cache are referenced via a Resident Object Table Handle (ROTHandle). Once a PCopyStub has been promoted to the object cache, it will be referenced via a ROTHandle. Also, since the ROTHandle has space in its header for a number of flags, a PCS flag can be set, as a shortcut for indicating that this is a PCopyStub object.

C.1.4 Residency check on persistent, non-resident PCopyStub

Once a PCopyStub has been made persistent, it may be accessed during the same or subsequent program executions. Step four of figure C.1 illustrates the stages through which a PCopyStub handle will go, from being accessed while not in memory at all, to faulting the remote object that it represents. The residency check done on access to a Java object detects when a PID is held for a non-resident, persistent object, instead of a reference to

the in-memory object itself. A local fault is triggered on the object corresponding to that PID, which transfers it from disk to object cache. Once a `ROTHandle` holds the `PCopyStub` in memory, the next stage is to trigger a remote fault to obtain the object that it represents. As described in section C.1.2, this should return a newly-created local copy of the remote object. To provide direct access to this object from this point in application execution, the object and methods fields of the `PCopyStub`'s ROTHandle are set to the object and methods of the newly-remote-faulted object and the methods of its class respectively. The `PCS` flag still remains set though, so that the objects may be handled correctly during promotion, as described in the next section.

C.1.5 Promotion of a remote-faulted object

If the `PCopyStub` for a remote-faulted object has never been persistent, then promotion of the remote-faulted object is treated as a normal object promotion. However, if the `PCopyStub` for a newly-remote-faulted object was persistent before this remote fault took place, this makes promotion of the remote-faulted object somewhat more complicated. This is to take into account the fact that several objects already in the persistent store may hold references to the `PCopyStub`, and thus, implicitly, to the newly-remote-faulted object which is about to be made persistent. To deal with this complication, although the newly-faulted object will continue to be reachable directly through the `PCopyStub`'s ROTHandle in memory, it will be reachable indirectly through the `PCopyStub` object on disk. The transition of the newly-remote-faulted object during promotion, from direct references in its `PCopyStub` ROTHandle, to reference via the `PCopyStub` object's `tmp` field, is illustrated in step five of figure C.1. To reinstate the persistent `PCopyStub` object, this is local-faulted from the persistent store first. The `ROTHandle` that referenced the newly-remote-faulted object, is set to reference its `PCopyStub` object and class methods once more. Then the `tmp` field of the `PCopyStub` object is set to reference a newly-created `JHandle`, set to reference the remote-faulted object and class methods. The `tmp` field of the `PCopyStub` is then marked as updated to ensure that the newly-remote-faulted object will be promoted to disk, now reachable from its `PCopyStub`.

C.1.6 Residency check on persistent, remote-faulted object

Once a remote-faulted object has been promoted to the persistent store, referenced via its `PCopyStub`, subsequent residency checks made on the `PCopyStub` will fault the persistent, remote-faulted object from disk to memory, as illustrated in step six of figure C.1.

No object, apart from the `PCopyStub` itself, should hold a PID for its remote-faulted object.

Any reference to the remote-faulted object should always go through its `PCopyStub` and therefore should only have the `PCopyStub`'s PID. Thus, a residency check logically made on the remote-faulted object will initially cause a local fault of the `PCopyStub`. The residency-checking code will detect that the `PCopyStub` has a non-null `tmp` field, holding a reference to the persistent, remote-faulted object. This will then trigger a local fault of the object in `PCopyStub->tmp`. The object and methods of the remote-fault object will then be installed in the fields of the `PCopyStub`'s ROTHandle so that the remote-fault object can then be accessed directly in memory.

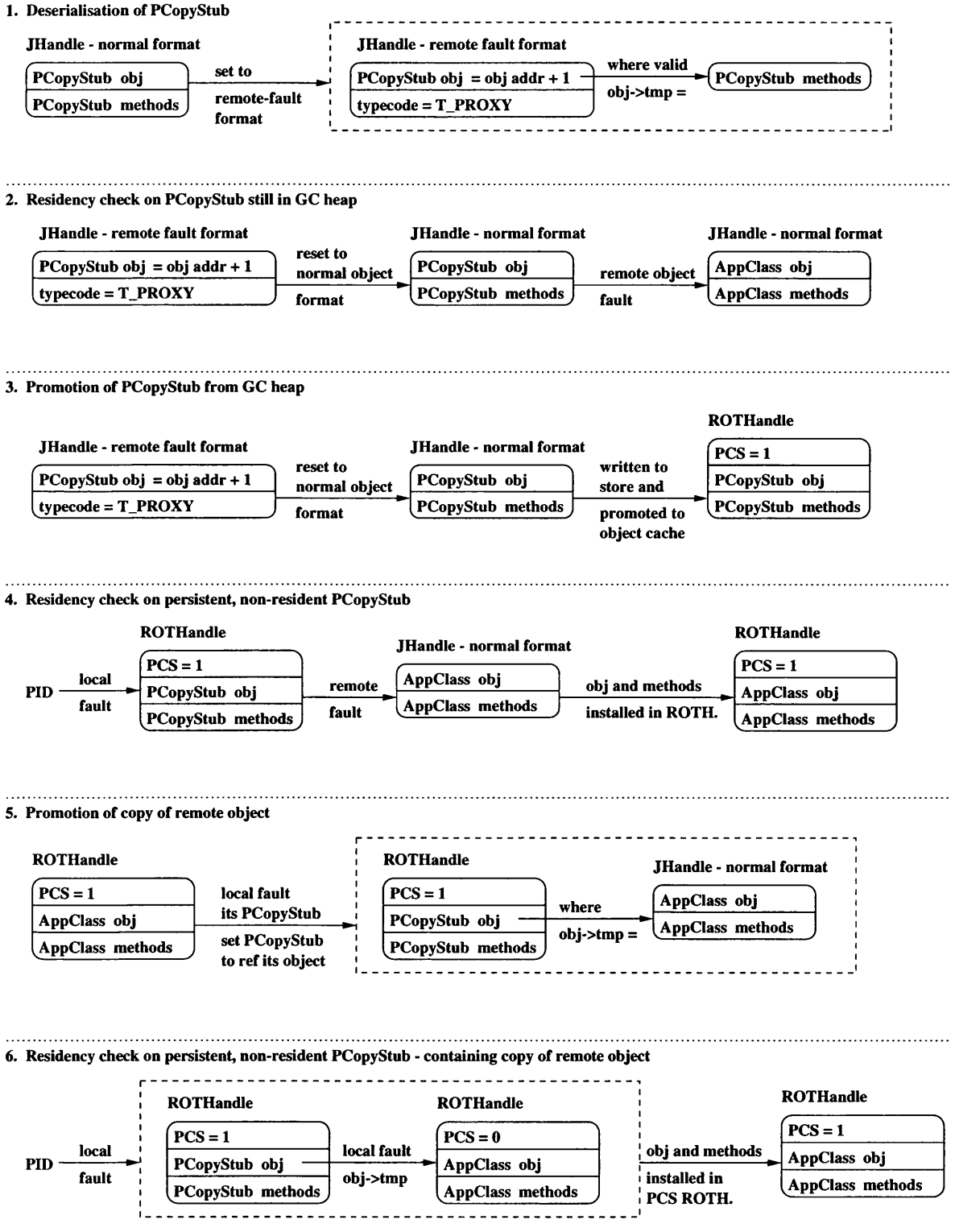

Figure C.1: Formats of PCopyStub/corresponding object copy handles during use

Trademarks

Glossary

application lease A lease, in the form of a time limit or duration of time, set to limit remote access to objects to within the scope of an application's lifetime.

application lifetime The lifetime of a distributed application is the time for which a group of distributed application programs run until the application is completed; this run may span multiple process executions, persistent store shutdowns and restarts.

distributed application Group of cooperating programs running as processes on a number of distributed machines.

externality An entity which is external to the persistent system e.g. a file, socket or thread.

JDK Java Development Kit.

JOS Java Object Serialization. Used in Java RMI for the marshalling and unmarshalling of parameters to remote method calls.

JVM Java Virtual Machine.

lease duration Period of time remaining until an application lease limit.

lease limit Time limit on an application lease.

object graph Transitive closure of all objects reachable directly and indirectly from a given root object.

object passing Passing objects as parameters or return values of calls made between two distributed processes, usually by reference or by copy.

object copying Passing objects by copy as parameters or return values of calls made between two distributed processes. This may involve copying some or all of the graph of objects reachable from the given object parameter.

orthogonal persistence Integration of data management and programming language where persistence is orthogonal to type, persistence independence is supported and a simple persistence identification mechanism (such as persistence by reachability) is used. See section 2.

OPJ Orthogonal Persistence for Java.

Orthogonal Persistence for Java Specification upon which the PJama implementations are based [JA99].

persistence by reachability The mechanism used in PJama to identify the objects to persist beyond the program execution in which they are created. An object registered by name using the PJama API is treated as a root of persistence. An object persists if it becomes reachable, directly or indirectly, from a persistent root.

persistence reachable An object is persistence reachable if it becomes persistent by reachability.

PJActionHandler PJama API class, instances of which are used to manage, at key points in the execution of a program over a persistent store, the state of objects that may be viewed as persistent by the application but which need special handling. See section 2.3.

PJama Implementation of Orthogonal Persistence for Java (aka PJava).

pjama0.5.7.13 Release of PJama based on JDK version 1.1.7. All PJama releases up to and including this version are based on a JDK version 1.1.x.

pjama0.5.20.0 Release of PJama based on JDK version 1.2 FCS. All PJama releases from this version upwards are based on JDK version 1.2.x.

pjama1.6.4 Release of second generation implementation of PJama based on JDK version 1.2. This second generation implementation has a simpler API and more scalable store implementation (Sphere) than previous releases.

PJama Project Collaboration between the University of Glasgow in Scotland and Sun Microsystems Laboratories in California, USA.

PJava Original name for implementation of Orthogonal Persistence for Java. Now known as PJama.

PJRMI Remote Method Invocation for PJama. See section 3.

PJVM PJama Virtual Machine: a JVM with modifications for support of orthogonal persistence.

remotely-invokable object Implementation of an interface whose methods can be called remotely.

RMI Registry Provided as part of standard RMI, it is a well-known service supporting look-up by name of remotely-invokable objects available on the Registry's host.

standard RMI Java RMI as implemented in an official release of Java from Sun Microsystems Inc.

stub An object which represents a remotely-invokable object remotely (aka proxy).

VM Virtual Machine. Used in this dissertation as a general term covering both Java Virtual Machine and PJama Virtual Machine.

Bibliography

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, 1983.

[ADJ+96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, 25(4):68–75, December 1996.

[AJ00] Malcolm Atkinson and Mick Jordan. Improved Hash Coding Methods for Java. Technical report, Sun Microsystem Laboratories, 2000. In Preparation.

[AM95] M.P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.

[AM97] Mehmet Aksit and Satoshi Matsuoka, editors. *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*. Springer, June 1997.

[Arj99] Arjuna Solutions Limited. *JTSArjuna 1.2.4 Programmer's Guide Volume 2: Using AIT*, 1999. Downloadable from http://www.arjuna.com.

[Atk96] M. Atkinson. Personal communication on experiences with DPS-algol, 1996.

[BC96] Krishna A. Bharat and Luca Cardelli. Migratory Applications. Technical report, DEC SRC, February 1996. SRC Research Report 138.

[BDF+00] S. J. Baylor, M. Devarakonda, S. J. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java Server Benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000.

[Bec99] Dan Becker. Design networked applications in RMI using the Adapter design pattern. *JavaWorld*, 4(5), May 1999.
http://www.javaworld.com/javaworld/jw-05-1999/jw-05-networked.html.

[Bel99] Beth Belton. Internet generated $301 billion last year. USA Today newspaper, 10 June, page 01A, 1999. Available in the USA Today Archives: http://usatoday.elibrary.com/s/usatoday.

[BEN+93] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed Garbage Collection for Network Objects. Technical Report 116, Systems Research Center, Digital Equipment Corporation, Palo Alto, December 1993.

[Ber91] S. Berman. *P-Pascal: A Data-Oriented Persistent Programming Language.* PhD thesis, University of Capetown, 1991.

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[Bla98] S.M. Blackburn. *Persistent Store Interface: A foundation for scalable persistent system design.* PhD thesis, Australian National University, Canberra, Australia, August 1998.

[BN84] Andrew Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. *Proceedings of the 14th ACM Symposium on Operating System Principles*, 27(5):217–230, December 1993.

[Car94] Luca Cardelli. Obliq: A Language with Distributed Scope. Technical report, Digital, June 1994. SRC Research Report 122.

[CBGM98] G. Canals, C. Bouthier, C. Godart, and P. Molli. Tuamotu : a Distributed Framework for Supporting Enterprise Projects. In *Proceedings of Colloque International sur les NOuvelles TEchnologies de la REpartition (NOTERE'98)*, Montréal, Québec, Canada, Oct 1998. Editions CRIM. http://www.iro.umontreal.ca/NOTERE.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 12–21. ACM Press, 1993.

[CMG98] G. Canals, P. Molli, and C. Godart. TuaMotu: Supporting Telecooperative Engineering Applications Using Replicated Versions. Internet-based GROupware for

User Participation in product development (IGROUP'98), Seattle, Washington, USA, November 1998.

[DA97] Laurent Daynès and Malcolm Atkinson. Main-Memory Management to support Orthogonal Persistence for Java. In Mick Jordan and Malcolm Atkinson, editors, *Proceedings of the Second International Workshop on Persistence and Java^{tm} (PJW2)*, number SMLI TR-97-63 in Sun Technical Report series, Half Moon Bay, California, August 1997.

[DA99] Misha Dmitriev and Malcolm Atkinson. Evolutionary Data Conversion in the PJama Persistent Language. In *Proceedings of the 1st ECOOP Workshop on Object-Oriented Databases*, Lisbon, Portugal, June 1999.

[DAV97] L. Daynès, M.P. Atkinson, and P. Valduriez. Customizable Concurrency Control for Persistent Java. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*, Data Management Systems, chapter 7. Kluwer Academic Publishers, Boston, 1997.

[Day00] L. Daynès. Implementation of Automated Fine-Granularity Locking in a Persistent Programming Language. *Software – Practice and Experience*, 30:1–37, 2000. To appear.

[DdBF+94] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan. Grasshopper: An Orthogonally Persistent Operating System. *Computing Systems*, 7(3):289–312, 1994.

[Dmi98] Misha Dmitriev. The First Experience of Class Evolution Support in PJama. In Morrison et al. [MJA98].

[DRV91] A. Dearle, J. Rosenberg, and F. Vaughan. A Remote Execution Mechanism for Distributed Homogeneous Stable Stores. In *Proceedings of the Third International Workshop on Database Programming Languages: Object Models and Languages*, pages 125–138, Nafplion, Greece, 1991. Morgan Kaufmann.

[dS96] M. Mira da Silva. *Models of Higher-order, Type-safe, Distributed Computation over Autonomous Persistent Object Stores.* PhD thesis, University of Glasgow, December 1996.

[dSA96] M. Mira da Silva and M. Atkinson. Higher-order Distributed Computation over Autonomous Persistent Stores. In *Proceedings of The Seventh International Workshop on Persistent Object Systems*, Cape May, New Jersey, USA, May 1996.

[dSAB96] M. Mira da Silva, Malcolm P. Atkinson, and A. P. Black. Semantics for Parameter Passing in a Type-complete Persistent RPC. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, pages 411–419, Hong Kong, May 1996. IEEE Computer Society.

[ECO00] ECOO. Environnements pour la coopération. WWW site for ECOO project developing CSCW support, 2000. http://www.loria.fr/equipes/ecoo/english/index.html.

[ED97] Huw Evans and Peter Dickman. DRASTIC: A Run-time Architecture for Evolving, Distributed, Persistent Systems. In Aksit and Matsuoka [AM97], pages 243–275.

[ED99] Huw Evans and Peter Dickman. Zones, Contracts and Absorbing Change: An Approach to Software Evolution. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '99)*, volume 34 of *SIGPLAN Notices*, pages 415–434, Denver, Colorado, USA, October 1999. ACM.

[EJB99a] Enterprise JavaBeans Specification 1.1 Introduces New and Exciting Features. Sun Microsystems Inc, November 1999. http://java.sun.com/products/ejb/newspec.html.

[EJB99b] Sun Microsystems Inc. *Enterprise JavaBeans Specification v1.1*, final release edition, December 1999.

[EJB99c] Sun Microsystems Inc. *Enterprise JavaBeans^{TM} Developer's Guide*, beta release edition, 1999.

[ES98] Huw Evans and Susan Spence. Porting a Distributed System to PJama: Orthogonal Persistence for Java. In Morrison et al. [MJA98].

[Eva99] H. Evans. Why Object Serialization is Inappropriate for Providing Persistence in Java. Technical report, University of Glasgow, UK, 1999. *In Preparation.*

[FD93] A. Farkas and A. Dearle. Octopus: A Reflective Mechanism for Object Manipulation. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages: Object Models and Languages*, pages 50–64, Manhattan, New York City, USA, 1993. Springer-Verlag.

[For00] The Forest Project. Sun Microsystems Laboratories, 2000. http://www.sun.com/research/forest.

[FS98] Paulo Ferreira and Marc Shapiro. Modelling a Distributed Cached Store for Garbage Collection. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming, 12th European Conference*, Lecture Notes in Computer Science, pages 234–259, Brussels, Belgium, July 1998. Springer.

[FSB+98] Paulo Ferreira, Marc Shapiro, Xavier Blondel, Olivier Fambon, Joâo Garcia, Sytse Kloosterman, Nicolas Richer, Marcus Roberts, Fadi Sandakly, George Coulouris, Jean Dollimore, Paulo Guedes, Daniel Hagimont, and Sacha Krakowiak. PerDiS: design, implementation, and use of a PERsistent DIstributed Store. Technical Report QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, QMW, CSTB, INRIA and INESC, October 1998.

[FT98] Dominant web role forseen. page 2, FT-IT Review, Financial Times Newspaper, 5 November, 1998.

[GC89] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. *SIGOPS Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 23(5):202–210, 1989.

[Gem96] GemStone Systems, Inc. *GemBuilder for VisualWorks, GemStone Version 5.0*, July 1996.

[Gem98a] GemStone Systems, Inc. *GemStone/J with VisiBroker Programming Guide, Version 1.1*, March 1998.

[Gem98b] GemStone Systems, Inc. *GemStone/J™ Distributed JavaBeans™ Programming Guide, Version 1.1*, March 1998.

[Gem99] GemStone Systems, Inc. GemStone/J™ 3.0: the Secure Integrated Application Platform for Internet Commerce. Product Overview published by GemStone Systems, Inc. on WWW, 1999. http://www.gemstone.com/products/j/overview.pdf.

[GGM96] Rachid Guerraoui, Benoît Garbinato, and Karim Mazouni. Lessons from Designing and Implementing GARF. In *Object Oriented Parallel and Distributed Computing*, LNCS. Springer Verlag, 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Han99a] Øyvind Hanssen. Personal communication on experience with his framework for policy bindings, 1999.

[HAN99b] Richard Hayton and the ANSA Team. *FlexiNet Architecture*, February 1999. Part of the ANSA Architecture for Open Distributed Systems Initiative.

[HE99] Øyvind Hanssen and Frank Eliassen. A Framework for Policy Bindings. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, September 1999. IEEE Press.

[HMB90] Antony L. Hosking, J. Eliot B. Moss, and Cynthia Bliss. Design of an Object Faulting Persistent Smalltalk. Technical report, University of Massachusetts, May 1990. COINS Technical Report 90-45.

[HPM93] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A Flexible Base for Distributed Programming. Technical report, Sun Microsystems Laboratories, April 1993. SMLI TR-93-13.

[II00] The Internet Economy Indicators^{TM} indicators report. University of Texas Center for Research in Electronic Commerce, 2000. http://www.internetindicators.com.

[JA98] Mick Jordan and Malcolm Atkinson. Orthogonal Persistence for Java - a Mid-term Report. In Morrison et al. [MJA98].

[JA99] Mick Jordan and Malcolm Atkinson. Orthogonal Persistence for the Java^{tm} Platform - Draft Specification. Technical report, Sun Microsystems Inc., 1999. In Preparation. Available from http://www.sun.com/research/forest/COM.Sun.Labs.Forest.doc.opjspec.abs.html.

[Jap00] Robert Japp. Personal communication on the performance of PJama on the Sphere architecture, 2000.

[JL99] Sun Microsystems Inc. *Jini Distributed Leasing Specification*, January 1999. Jini System Software 1.0 Specifications.

[JOP96] JOP (Java Object Persistence), Alpha release 0.4a, 15 Sep 1996. David Rothwell, 1996. http://www.magna.com.au/~davidr/.

[Jor99] D. Jordan. Serialisation is not a Database Substitute. *Java^{tm} Report*, pages 68–79, July 1999.

[JOS97] Java Object Serialisation Specification, Draft Revision 1.3. Sun Microsystems Inc, 1997.

[JOS98] Java^{tm} Object Serialisation Specification, jdk^{tm} 1.2. Sun Microsystems Inc, November 1998. Documentation supplied with Release of the JDK 1.2 FCS.

[JV97] Mick Jordan and Michael L. Van De Vanter. Modular System Building with Java(tm) Packages. In *Eighth Conference on Software Engineering Environments*, pages 155–163, Cottbus, Germany, April 1997.

[KDM99] Setrag Khoshafian, Surapol Dasananda, and Norayr Minassian. *The Jasmine Object Database: Multimedia Applications for the Web.* Morgan Kaufmann, 1999.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Aksit and Matsuoka [AM97], pages 220–242.

[LAC+96] Barbara Liskov, Atul Adya, Miguel Castro, Mark Day, Sanjay Ghemawat, Robert Gruber, Umesh Maheshwari, Andrew C. Myers, and Liuba Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 318–329, Quebec, Canada, June 1996. ACM Press.

[LCSA99] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing Persistent Objects in Distributed Systems. In Rachid Guerraoui, editor, *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 230–257. Springer, June 1999.

[Lin99] Kevin C.F. Lew Kew Lin. *Orthogonal Persistence, Object-orientation and Distribution.* PhD thesis, University of Adelaide, September 1999. Submitted.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[LK99] Cristina Lopes and Gregor Kiczales. Aspect-Oriented Programming with AspectJ. Notes for Tutorial #2, November 1999. Conference on Object-Oriented Programming, Systems, Languages and Applications 1999 (OOPSLA99).

[LS96] M.C. Little and S.K. Shrivastava. Using Application Specific Knowledge for Configuring Object Replicas. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 169–176, Annapolis, Maryland, May 1996. IEEE Computer Society Press.

[LS99a] M.C. Little and S.K. Shrivastava. A method for combining replication with cacheing. In *Proceedings of the IEEE International Workshop on Reliable Middleware Systems (WREMI99)*, Lausanne, Switzerland, October 1999. IEEE.

[LS99b] M.C. Little and S.K. Shrivastava. Implementing high availability CORBA applications with Java. In *Proceedings of the IEEE Workshop on Internet Applications*, San Jose, California, June 1999. IEEE.

[MBC+96] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby, and D.S. Munro. Napier88 Reference Manual (Release 2.2.1). Technical report, University of St Andrews, 1996.

[MCC+99] R. Morrison, R.C.H. Connor, Q.I. Cutts, G.N.C. Kirby, D.S. Munro, and M.P. Atkinson. The Napier88 Persistent Programming Language and Environment. In M.P. Atkinson and R. Welland, editors, *Fully Integrated Data Environments*. Springer-Verlag, 1999.

[MJA98] Ron Morrison, Mick Jordan, and Malcolm Atkinson, editors. *Proceedings of the Third International Workshop on Persistence and Java*, Tiburon, CA, September 1998. Morgan Kaufmann.

[ML97] Umesh Maheshwari and Barbara Liskov. Collecting Distributed Garbage Cycles by Back Tracing. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 97)*, pages 239–248, Santa Barbara, California, USA, August 1997. ACM.

[MOM98] Jishnu Mukerji and OMG. *CORBA Core Chapter revisions from RFP on Objects-by-Value*, 1998. ftp://ftp.omg.org/pub/orbrev/drafts/obv-java-jm-0730-2.3-rtf.pdf.

[Mos90a] J. Eliot B. Moss. Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach. In Hull, Morrison, and Stemple, editors, *Second International Workshop on Database Programming Languages*, pages 358–374, Gleneden Beach, OR, June 1990. Morgan Kaufmann.

[Mos90b] J. Eliot B. Moss. Design of the Mneme Persistent Object Store. *ACM Trans. on Information Systems*, 8(2):103–139, April 1990.

[NPS95] B. Noble, M. Price, and M. Satyanarayanan. A programming interface for application-aware adaption in mobile computing. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, Ann Arbor, Michigin, USA, April 1995.

[NSN+97] B. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.

[OH98] Robert Orfali and Dan Harkey. *Client/Server Programming with Java™ and CORBA*. Wiley, second edition, 1998.

[OMG] Object Management Group. http://www.omg.org.

[OMG98] CORBAservices: Common Object Services Specification. Object Management Group, Inc. Publications and downloadable as document formal/98-12-09 from http://www.omg.org/corba, December 1998.

[OMG99a] CORBA Success Stories. http://www.corba.org, 1999.

[OMG99b] Persistent State Service 2.0. Object Management Group, Recently Adopted Specifications downloadable as document orbos/99-07-07 from http://www.omg.org/techprocess/meetings/schedule/tech2a.html, August 1999.

[OMG99c] The Common Object Request Broker: Architecture and Specification, Version 2.3.1. Object Management Group, Inc. Publications and downloadable as document formal/99-10-07 from http://www.omg.org/corba, October 1999.

[Ora99] Oracle. Oracle 8i: Features Overview Document. http://www.oracle.com/database/documents/o8i_new_features_summary_fo.pdf, 1999.

[OS98] Michael Otto and Norbert Schuler. Persistente softwaretechnische archive zur kooperationsuntersttzenden verwaltung behlterartiger materialien. Master's thesis, University of Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Universitt Hamburg, Vogt-Klln-Strae 30, 22527 Hamburg, oct 1998. Betreuung: Dr. Guido Gryczan, Report written on hierarchical archive implemented using PJama for student project written up as a bachelor thesis.

[PAD98a] T. Printezis, M. P. Atkinson, and L. Daynès. The Implementation of Sphere: a Scalable, Flexible, and Extensible Persistent Object Store. Technical Report TR-1998-46, University of Glasgow, July 1998.

[PAD+98b] Tony Printezis, Malcolm Atkinson, Laurent Daynès, Susan Spence, and Pete Bailey. The Design of a new Persistent Object Store for PJama. In Morrison et al. [MJA98].

[PAJ99] T. Printezis, M. P. Atkinson, and M. J. Jordan. Defining and Handling Transient Data in PJama. In *Proceedings of DBPL'99*, Kinlochrannoch, Scotland, September 1999.

[PJR98] PJama: implementation of Orthogonal Persistence for Java, Release 0.5.7.13. The PJama Project. Sun Microsystems Laboratories and University of Glasgow, December 1998.

[PJR99] PJama: implementation of Orthogonal Persistence for Java, Release 0.5.20.2. The PJama Project. Sun Microsystems Laboratories and University of Glasgow, 1999.

[PJR00] PJama: implementation of Orthogonal Persistence for Java, Release 1.6.4 (for JDK 1.2). The PJama Project. Sun Microsystems Laboratories and University of Glasgow, March 2000. http://www.sun.com/research/forest/opj.main.html.

[PNC98] John Potter, James Noble, and David Clarke. The Ins and Outs of Objects. In *Australian Software Engineering Conference (ASWEC98)*, Adelaide, Australia, November 1998.

[POE98] POET: Persistent Objects and Extended Database Technology. POET Software, 1998. http://www.poet.com.

[Pri00a] T. Printezis. *Management of Long-Running High-Performance Persistent Object Stores*. PhD thesis, Department of Computing Science, University of Glasgow, Scotland, 2000.

[Pri00b] T. Printezis. Personal communication on weak references and disk garbage collection, 2000.

[RC89] J. Richardson and M.J. Carey. Persistence in the E Language: Issues and Implementation. *SPE*, 19(12):1115–1150, 1989.

[RE98] Rosemary Rock-Evans. *DCOM Explained.* Digital Press, 1998.

[RMI98] Java^{TM} Remote Method Invocation Specification, Revision 1.50, JDK 1.2, October 1998. Documentation supplied with JDK 1.2 FCS.

[Sch77] Joachim W. Schmidt. Some high level language constructs for data of type relation. *TODS*, 2(3):247–261, 1977.

[Ses98] Roger Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects.* Wiley, 1998.

[SKW92] Vivek Singhal, Sheetal Kakkad, and Paul Wilson. Texas: An Efficient Portable Persistent Object Store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992.

[Sun99] Sun Microsystems Inc. Sun on the Net: How Sun saves money and improves service using Internet technologies. http://www.sun.com/960101/feature1, February 1999.

[USC98] US Census Bureau. *TIGER/Line® Files 1998, Technical Documentation*, July 1998. http://www.census.gov/geo/www/tiger/index.html.

[vRBM96] R. van Renesse, K. Birman, and S. Maffeis. Horus, a Flexible Group Communication System. *Communications of the ACM*, April 1996.

[Wai88] F. Wai. *Distributed Concurrent Persistent Programming Languages: An Experimental Design and Implementation.* PhD thesis, University of Glasgow, April 1988.

[Wai89] Francis Wai. Distributed PS-algol. In John Rosenberg and Davis Koch, editors, *Persistent Object Systems, Proceedings of the Third International Workshop (POS3)*, Workshops in Computing, pages 126–140, Newcastle, New South Wales, January 1989. Springer.

[WB95] Girish Welling and B. R. Badrinath. Mobjects: Programming Support for Environment Directed Application Policies in Mobile Computing. In *Proceedings of the ECOOP'95 Workshop on Mobility and Replication*, Aarhus, Denmark, August 1995.

[WB97] Girish Welling and B. R. Badrinath. A Framework for Environment Aware Applications. In *17th International Conference on Distributed Computing Systems (ICDCS)*, pages 384–391, Baltimore, Maryland, USA, May 1997.