Models of Higher-order, Type-safe, Distributed Computation over Autonomous Persistent Object Stores

Miguel Leitão Bignolas Mira da Silva

A thesis submitted to the Faculty of Science, University of Glasgow
for the degree of Doctor of Philosophy

December 1996

Abstract

A remote procedure call (RPC) mechanism permits the calling of procedures in another address space. RPC is a simple but highly effective mechanism for interprocess communication and enjoys nowadays a great popularity as a tool for building distributed applications. This popularity is partly a result of their overall simplicity but also partly a consequence of more than 20 years of research in transparent distribution that have failed to deliver systems that meet the expectations of real-world application programmers.

During the same 20 years, persistent systems have proved their suitability for building complex database applications by seamlessly integrating features traditionally found in database management systems into the programming language itself. Some research effort has been invested on distributed persistent systems, but the outcomes commonly suffer from the same problems found with transparent distribution.

In this thesis I claim that a *higher-order persistent RPC* is useful for building distributed persistent applications. The proposed mechanism is: *realistic* in the sense that it uses current technology and tolerates partial failures; *understandable* by application programmers; and *general* to support the development of many classes of distributed persistent applications.

In order to demonstrate the validity of these claims, I propose and have implemented three models for distributed higher-order computation over autonomous persistent stores. Each model has successively exposed new problems which have then been overcome by the next model. Together, the three models provide a general yet simple higher-order persistent RPC that is able to operate in realistic environments with partial failures.

The real strength of this thesis is the demonstration of realism and simplicity. A higher-order persistent RPC was not only implemented but also used by programmers without experience of programming distributed applications. Furthermore, a distributed persistent application has been built using these models which would not have been feasible with a traditional (non-persistent) programming language.

Acknowledgements

First of all I'd like to thank Malcolm Atkinson, my supervisor, for all his support, advice and criticism. I am specially indebted for the many discussions and text revisions of my reports and papers, and particularly the dissertation. Malcolm is an example of hard work and has provided an excellent research environment in Glasgow. I am also grateful for his support of my research trips which gave me the opportunity to meet so many interesting people.

My research colleagues in Glasgow also deserve my thanks. Although it's impossible to acknowledge them all, I'd like to mention Peter Dickman, who played the difficult role of second supervisor. Other colleagues that I'd like to thank include Paul Philbrow, Huw Evans, Susan Spence, Laurent Daynès, Darryn Lavery, João Lopes, Tony Printezis, Stewart Macneill, Quintin Cutts and Peter Larsson.

Thanks are also due to past and present members of the Persistent Programming Research Group in St. Andrews, who have built a robust implementation of Napier88. Of my research friends around the world, I'd particularly like to thank Andrew Black and Bernd Mathiske, with whom I discussed many of the topics in this thesis, and Dag Sjøberg for all the advice and support.

* * *

But life is not only work, and these last three years would not have been possible without the support of all my friends. I'd like to thank Jorge Pedroso, Ana Vasconcelos, Cristina Gomes, Pedro Carvalho, Ana Fonseca, Manuella Coelho, Margarida Dias and so many others in Portugal; Eva, Paula, Ana Roque and Teresa in Glasgow; Monica in the States; and Patricia in Germany.

Last but not the least, I'd like to thank my family for the support, understanding, and friendship that made this journey possible.

* * *

This thesis is based on research work partially supported by JNICT, the Portuguese Council for Scientific and Technological Research.

Table of Contents

List of Figures

List of Examples

List of Tables

Chapter 1

Introduction

This thesis is concerned with identifying an architecture that improves on existing models of inter-process communication to facilitate the construction and maintenance of geographically distributed applications that manipulate large amounts of complex data.

Two existing research areas deal directly with different aspects of this problem.

- *Persistent systems* store and manipulate complex data irrespective of their life time, typically based on a programming language.
- *Distributed systems* provide models, algorithms and mechanisms that permit geographically distributed programs to collaborate in order to behave like a single application.

This introductory chapter describes the main features of persistence and distribution, and why the problems posed by the combination of these have not been solved by the technology currently available. We then present the thesis statement and give an overall picture of the three models proposed in this thesis.

1.1 Research Context

In order to understand the problems posed by the integration of persistence and distribution, we first give an introduction to each of these research areas.

1.1.1 Persistent Systems

There are many ways to implement a persistent system, but in this dissertation we will concentrate on those based in programming languages.

An *orthogonal persistent programming language* makes no distinction between short-term and long-term data; all data of all types is kept by the system as long as it is useful for the application [ABC+83, AM95].

Persistence is usually defined by reachability from one or more persistent roots [Bro88, BR90, Mun93]. Objects not reachable, directly or indirectly, from a persistent root are candidates for garbage collection. Internally, the system needs to save long-lived objects on disk or other non-volatile media, but this is completely transparent to application programmers.

Main Features of Persistent Systems

Orthogonal persistence has a number of interesting consequences. Most of these were introduced to benefit application programmers and—ideally, at least—*should be preserved in a distributed environment.* However, as we will see during this dissertation, they also introduce new problems for system implementors.

- *Orthogonal persistence*—This means persistence is orthogonal to other characteristics of the object, such as its type. Thus *any object of any type has the right to become persistent*, including simple types, constructed types (records, unions, arrays) and data structures of any complexity (e.g., pieces of source code and their relationships in a CASE tool).

- *Type-safety*—It is good language design to enforce type-safety in order to prevent application programmers from making type errors. Type-safety is even more important in a persistent language to maintain the integrity of the persistent store. Together, orthogonal persistence and type-safety enforce *long-term referential integrity.* This means that C-like "dangling pointers" never occur in a persistent language and this guarantee extends to long-lived data.

- *Higher-order*—Procedures are a useful abstraction in any programming language, but there are many advantages in promoting them to full citizenship [AM85]. This means that procedures can be created at run-time, passed as arguments or returned as results in other procedures, and for an orthogonal persistent language it also means that *procedures can become persistent* like any other data object.

The Persistent Language Napier88

We use the persistent programming language Napier88 [MBCD89, MBC+94] for the research reported in this dissertation. Napier88 will be described extensively in chapter 3; for the time being, it only matters that Napier88 supports all the characteristics mentioned above.

1.1.2 Distributed Systems

A distributed system supports the execution of programs in separate address spaces—especially in different computers—while at the same time attempting to give the illusion that together they behave as a single (distributed) application. There are, however, many kinds of distributed system; this dissertation is specially concerned with the level of illusion that each system attempts to give to the application programmer using it.

For example, the World-wide Web is a huge, distributed application: while an end-user on a PC only starts a program (the browser), that program will talk to other programs (Web servers) all over the world to present the HTML pages [Wor96] requested by the user. In theory at least, the end-user does not need to know whether the information is coming from the local hard disk or from an IBM mainframe on the other side of the World.

In reality, however, users of a distributed application develop some conceptual model of the distribution. For example, most European users of the WWW are well aware that they get much better response when they use American sites in the morning, before American users start imposing their load. But many are unaware of various forms of caching (in Web proxies) that make some of that data more locally available, and probably most are unaware of the computers (gateways) that act as intermediaries when their requests are being answered.

From the Web example, we can infer that end-users will readily comprehend a rational and simple model of distribution. But they certainly do not wish to consider many of the details that application programmers *must* face when building and maintaining the distributed application. These application programmers must strive to present the rational and simple model to end-users.

It is a goal of this thesis to help application programmers in that task by developing models for distributed persistent applications that provide an appropriate abstraction of the real distributed system.

Main Features of Distributed Systems

The main characteristics of distributed systems that we would like to maintain in a persistent environment are autonomy and collaboration. Here we give only a first motivation for these; a more detailed analysis will appear in chapter 2.

The Web is an extreme example of autonomy. Each Web site—a Web server and its HTML pages—is free to add, change and remove pages, create new CGI scripts to generate HTML on-demand, and even choose a particular implementation of the Web server. The Web illustrates that autonomy is necessary for local management and evolution. Autonomy is also appropriate for a large number of sites because there is no need for centralized control.

However, when the Web is stressed to its limits, autonomy segregates providers and con-

sumers of information into categories and makes the distributed application behave more like a collection of sub-applications. For example, even though the Web is supposed to support standard HTML, many sites these days have specific versions for different classes of readers (HTML 2 or 3, with or without pictures, frames, *applets* and so on). Autonomy also creates problems with referential integrity between (even within) sites — the well-known "broken link" problem. Finally, autonomy permits different assumptions about data types, network bandwidth, user-interface, and so on to proliferate.

On the other hand, autonomy is only possible because the HTTP protocol used for communication between browsers and servers is a *de facto* standard. Thus, at the same time it suggests the need for autonomy, the Web also suggests the need for less autonomy and more collaboration.

In a real distributed application, a trade-off between autonomy and collaboration is needed.

One of the most challenging design issues is that *the trade-off is application dependent.* Sometimes the distributed application needs stronger collaboration between sites. At other times, more autonomy is needed. (Some examples will be given in chapters 2 and 3.) Probably even more often, both autonomy *and* collaboration are required.

One solution to this problem — that we choose to explore in this dissertation — is providing enough support to enable application programmers to select the right compromise. In order to achieve this goal, the support needs to be easy to understand and use, otherwise only programmers with distribution expertise would be able to use it.

Models of Distribution

In order to address this trade-off between autonomy and collaboration that exists in all distributed applications, research in distributed systems has been typically divided into sub-areas. Each sub-area is based on one of two models of distribution.

- *Federated model* — In this model *autonomy is preserved* because programs are for the most part independent and only talk to each other by means of an inter-process communication (IPC) mechanism. Application programmers need to understand the distributed semantics and use the IPC themselves to build the distributed application.
- *One-world model* — In this model (also called transparent distribution) the idea is to compromise on local autonomy to achieve *overall simplicity.* The system takes the responsibility for dealing with all (or at least most) aspects of distribution and attempts to hide some or all aspects of distribution from application programmers.

In this dissertation we consider large distributed persistent applications, rather than the typical client/server applications running on local area networks. A typical example is a hospital information system. In a hospital, computing resources are typically distributed

across a variety of *autonomous* computer systems, each managing a quasi-independent part of the hospital (accounting, patients, pharmacy, catering, operating theatres and so on).

These long-lived, large-scale applications are characterised by a number of technological issues (see sections 2.1 and 2.2): *autonomy* for evolution, protection, management, and performance; *fault-tolerance* for dealing with a high rate of failures, some of which may never recover; and *heterogeneity* to support legacy data and applications.

These characteristics make them unsuitable for the one-world model of distribution. This is the reason why the dissertation concentrates on the federated model and its IPC support.

1.1.3 Interprocess Communication (IPC)

One solution for building large distributed persistent applications that preserves autonomy, deals with partial failures and is suitable for heterogeneous environments is the federated model of distribution based on IPC. An IPC mechanism supports communication between independent programs and is a simple yet effective way of building this kind of application.

We have described earlier in this chapter the main features of persistence and distribution. Distributed persistent applications require a combination of these research areas, but we cannot afford to lose the main existing features of each area in our implementation of IPC.

1. *Type-safe*—Since locally a persistent language like Napier88 is strongly type-safe, it is highly desirable to maintain this guarantee when the computation extends to another program.

2. *Higher-order*—First-class procedures have already proven to be useful in traditional (non-persistent) programming languages. It has been shown that first-class procedures are even more useful in a *persistent* environment [AM85]. They will probably also be useful for building distributed persistent applications.

3. *Persistent*—It is important to take advantage of orthogonal persistence, persistence independence and persistence by reachability, but also to take into account the new problems introduced by this technology. The canonical example is the difficulty posed by large objects that reside in the database, as it may be difficult to copy their values as arguments in a remote procedure.

4. *Autonomy and collaboration*—They will both have to be implemented by application programmers with a balance dependent on the distributed application.

An IPC that preserves these fundamental characteristics is well positioned to take full advantage of persistence and distribution when building distributed persistent applications.

1.1.4 Limitations of Current IPC Technology

The current technology for building federated distributed application systems is the *Remote Procedure Call* (RPC) [BN84]. Examples of RPC systems include Sun/RPC [Sun93b] and a number of implementations of CORBA [OMG95]. Although some authors would claim CORBA offers a model of *distributed objects*, the technology behind all CORBA implementations is the same as RPC.

Most implementations of RPC have a number of well-known characteristics that can be compared with our list above of required features for a persistent IPC.

1. *Restricted data types as arguments*—Most RPC systems do not support interesting types such as procedures as arguments in remote calls. This weakness is perfectly acceptable if these types are not supported by the programming language from which the RPC system is used. However, it becomes a major limitation if the language does support first-class procedures and other rich types because application programmers are prevented from using the same abstractions for local and distributed computation.

2. *Not type-safe*—The same kind of reasoning applies to type-safety: since many RPC systems are built for unsafe languages like C, these systems do not give full guarantees concerning the type of their arguments. However, invalid arguments cannot be acceptable for large distributed applications in which many programs are written and then evolve autonomously.

3. *No support for persistence*—Most popular RPC systems were designed for traditional programming languages and so only take volatile data into account. In contrast, most real-world distributed applications make some use of persistent data, and many access large and complex databases. This means application programmers have to deal with three systems offering three different programming models: the database, the language and the IPC mechanism.

4. *Flexibility*—One of the advantages of RPC is that it is sufficiently flexible to permit application programmers to achieve the right balance between autonomy and collaboration as required for the particular distributed application being built.

There are also new problems introduced by persistence. The transitive closure of a persistent object is typically large (see section 5.2.4). However, most RPC systems pass arguments by copy and as a consequence large amounts of data will be transmitted in every remote call. This is both inefficient and unnecessary because some of this data will already exist remotely and some will simply not be accessed at all. Passing arguments by copy also creates multiple copies of the same objects in the remote environment, destroying sharing semantics.

Persistent programmers are used to a type-safe persistent programming environment with a rich type system that can include first-class procedures. After getting used to these features, programmers will not easily accept restrictions on the data types permitted as

arguments to remote procedures. They limit the kind of distributed applications that can be built, and, in particular, the extension of existing persistent applications towards distribution.

After consideration of the weaknesses of the traditional RPC systems with respect to persistence, we conclude it is not suitable for building distributed persistent applications.

Some of these limitations are finally being recognized. For example, an implementation of RPC for Java [AG96], called RMI [WRW96, RWW96], is both type-safe and is said to support any Java data type as an argument in a remote call (provided a serialisation method has been defined). In practice, objects belonging to the class *Thread* and all other objects that are implemented by the run-time system, like AWT objects, cannot (currently) be used. Another mechanism — based on special classes called *applets* — is needed to send *code* between Java programs. Finally, although there is some support for persistence in Java, *applets* cannot survive a program execution.

1.2 Thesis Statement

I claim that it is possible to design and build a *simple*, *general*, and *realistic* IPC that can be used by typical application programmers to construct and maintain distributed persistent applications without compromising too much on store autonomy.

- *Simple* — To be *understandable* by typical application programmers.
- *General* — To be *useful* for a variety of application categories.
- *Realistic* — To be *feasible* so it can be used for constructing real applications.

In order to demonstrate this claim I will propose, design, implement, employ and test three models that together address the main issues for higher-order, type-safe, distributed computation over autonomous persistent stores.

1.3 Models for Distributed Computation

The three models proposed in this dissertation are all extensions to the basic RPC model. RPC was chosen as the starting model because it provides a simple but flexible distributed programming model. In addition, RPC is a well-known paradigm, widely used for building distributed applications based on the federated model.

In this section we present an introduction to each of these models: persistent, type-safe RPC; migration by substitution; and persistent spaces. Each model solves a particular problem and reveals a number of others, that are in turn solved by the next model. (The problems revealed by persistent spaces are research issues for future work.)

1.3.1 Persistent Type-safe RPC

The first model described in this dissertation—also called Napier/RPC release 1.0—is a basic RPC mechanism that takes advantage of orthogonal persistence and is type-safe. This model is described in chapter 4.

Here we give one example of how we have used persistence. In contrast to most RPC implementations—that require an external "interface language" to define the remote procedure signature—Napier/RPC is based on *internal stub generation*. This permits new stubs to be created at run-time (a feature used later by persistent spaces).

Type-safety is implemented with capabilities and a binding service. Traditional type-checking at compilation-time does not apply because in general it is not possible to have access to all programs in a distributed application simultaneously.

Napier/RPC 1.0 was first described in a paper presented to the RIDE'95 Workshop on Distributed Object Management [MdS95a]. Since then, a failure model has been incorporated and its name has been changed to Napier/RPC 2.2 [MdS95b]. Later, Napier/RPC 2.2 became part of Glasgow Libraries [CAL+94]. Chapter 4 presents the original type-safe persistent RPC and an implementation (not described in the RIDE'95 paper).

1.3.2 Migration by Substitution

This second model was developed after we realized that the semantics for *migrating* the arguments is perhaps the most important design issue in a higher-order persistent RPC. (In this dissertation, the word *migration* will be used as a general term that comprises movement, copying, and replication of objects between programs.) The main reason is that procedures—especially those that live in the persistent store—have large transitive closures (see section 5.2.4). This characteristic extends to all objects that include procedures in their transitive closures.

Passing large objects by copying them to the remote procedure is not only unacceptably inefficient but also makes for poor usage of store space. In addition, it also spoils substructure sharing semantics over multiple arguments or multiple calls.

Many schemes to solve this problem have been proposed in the literature (see sections 3.3.6 and 5.3.1). They can all be classified into two basic models according to the "amount of information" that is carried when a remote procedure is called.

1. *Migrating by reference*—Only a reference to the local argument object (such as a globally unique object identifier) is passed to the remote procedure. Subsequent accesses to this object will then require *migrating the thread of execution to where the object resides.* Alternatively, a total or partial copy of the object can be performed on accesses to the object.

2. *Migrating by copy*—The value of the argument object is duplicated in the remote program when the procedure is called. When the application must cope with partial failures and recovery (as most commercial products have to) the system normally uses migration by copy because no further communication is required between calling the procedure and returning the result.

We aim to build *realistic* systems. This is the reason why *migration by copy was chosen as the basic model for passing parameters.* Adjustments to the basic semantics of copying can be made if strictly necessary, but only after evaluating very carefully how much autonomy is being lost.

In chapter 5 we propose a compromise between migrating by reference and by copy called *migration by substitution.* This novel parameter passing model lets application programmers define which local objects are to be substituted by equivalent remote objects. These objects *will not be copied* to the remote store, so we avoid the problem of maintaining the copies consistent with the original objects. As we will explain in chapter 5, there is also no need to maintain remote references to the original objects.

Migration by reference, by copy and by substitution were originally presented at the ICDCS'96 Conference on Distributed Computing Systems [MdSAB96]. In this dissertation we further discuss these models and the need for a compromise.

1.3.3 Persistent Spaces

The third model proposed in this thesis, described in detail in chapter 6, provides a simple and efficient mechanism for caching complex objects in distributed persistent applications.

The motivation for persistent spaces is that most objects in a certain class of distributed persistent application are stable for long periods of time. In this environment, a remote procedure call for every access to objects in another store is clearly inappropriate. Instead, programmers can use a *persistent space* to cache some popular objects locally and only make a remote call when the original object is updated to fetch the new value. The basic RPC mechanism can still be used to access large, rapidly changing or infrequently accessed objects.

The main contribution of persistent spaces is a two-phase mechanism to control the data flow between a publisher and its subscribers. A publisher store determines when a new version of an object becomes visible by putting that object into a persistent space. Any number of subscriber stores then choose when to request the new version.

Another contribution is a technique for refreshing local replicas on subscribers of very complex objects without requiring the entire object to be copied again; only those parts of the object that have changed are sent. This mechanism also saves disk space, especially for large persistent objects. (The argument that disk space is cheap these days does not hold because objects will just grow to fill up any free disk space.)

Persistent spaces were originally presented at the Seventh Workshop on Persistent Object Systems [MdSA96b]. This dissertation further motivates their use, presents an example application and describes an implementation in a persistent programming language.

1.4 Dissertation Structure

This dissertation is structured into eight chapters. The main contributions outlined in section 1.3 comprise chapters 4 to 6.

1. *Introduction* — This chapter.
2. *Survey of inter-process communication* — Distributed applications. Models of distribution. IPC design issues.
3. *Overview of persistence and RPC* — Orthogonal persistence. The Napier88 persistent programming language. Overview of RPC.
4. *Type-safe persistent RPC* — Design and implementation of a type-safe RPC mechanism that takes advantage of orthogonal persistence (see section 1.3.1 above).
5. *Extending object migration* — A survey on existing models for migrating objects between programs. Design and implementation of a new model called migration by substitution (see section 1.3.2 above).
6. *Persistent spaces* — Motivation, design, interface and implementation of a new model for sharing complex persistent objects between autonomous persistent programs (see section 1.3.3 above).
7. *Evaluation* — An example of a real distributed persistent application that makes use of all three models proposed. Performance measurements and discussion.
8. *Conclusion* — Summary and future work.

Chapter 2

Survey of Inter-process Communication

This chapter presents the context in which IPC works by first introducing distributed applications and then identifying the model of distribution to which this thesis contributes. This is followed by a section on high-level IPC design issues. The chapter will be used as a basis for describing more specific issues on persistent RPC design in chapter 3 and our research work in chapters 4 to 6.

2.1 Distributed Applications

A distributed application is composed of a number of components — sub-applications or programs — that interact with each other to achieve a common goal. Typically each component is represented by an operating system process executing in a computer; however, several components may share an address space, possibly for performance reasons.

2.1.1 Advantages

Research on distributed systems and applications is justified for a number of reasons. In this dissertation we are interested in both distribution and persistence, so Cheng's list of potential advantages of a distributed object-oriented database [Che93] is relevant in our context.

1. *Autonomy* — Users can enforce local policies such as database design, schema change, tuning, protection, back-up, and so on. Parts of the system can continue in operation despite network and other non-local failures.

2. *Performance*—Data and code can be placed where they are used, thus reducing communication costs. Each machine does its own processing instead of overloading a centralized server. Data replication can further increase performance.

3. *Availability*—When the network or a machine fails the application as a whole can usually continue operating, although eventually in a degraded manner. Replication can increase availability by maintaining local copies of data.

4. *Expandability*—Also called scalability, means that the system can grow incrementally by adding new programs, processing power and storage space. This is crucial as most applications are already built and companies cannot afford to re-build and replace them in a single step. Distribution is a necessary but not sufficient condition to overcome this problem.

5. *Shareability*—It is necessary to support sharing of data between applications in a computing environment that would be distributed anyway. This happens either because the organisation is geographically dispersed (and departments and groups are autonomous and develop their own solutions) or simply for historical reasons [Bir88, AE90].

6. *Economy*—It may be cheaper to buy a number of smaller machines than a single high-performance server. (But it may also be more expensive to maintain a larger number of smaller machines, see section 2.1.2.)

Perhaps the most important reason is the habits and wishes of people—they like to use data and programs they control. In consequence, more data now resides on the desktop than in central repositories. But these people have to cooperate and so their systems must support this cooperation. Consequently, from the list above, we emphasize both *autonomy* and *shareability*.

Autonomy also permits *heterogeneity*, fundamental for integrating different computing systems and accessing existing code and databases. (Difficulties anticipated for the integration of existing information systems have been given as one reason for not merging large companies.) Distributed information systems may help to make organisations more flexible.

2.1.2 Disadvantages

While they are not so often cited, distribution also has a number of disadvantages that slow or limit its adoption.

1. *Complexity*—Distributed systems are often more complex than centralized ones due to the requirement to cope with heterogeneity, partial failures and scale. Security, recovery, and management, all become harder and thus more expensive (see last item) as a result of distribution.

2. *Lack of Experience*—Distribution is still a new approach in many real-world application domains. Sometimes products are untried and application programmers inexperienced. This lack of experience of how to build and operate distributed systems becomes even more problematic because of their inherent complexity.

3. *Costs*—Usually neglected, CPU and communication costs are often an important part of distribution, especially in large, geographically distributed systems. While these may be ignored for research purposes or during initial evaluations, they are fundamental in an operational environment. Another source of operating costs is the system administrator's time needed for integration, management, upgrades, security, backup and recovery of the distributed system, which are all much simpler in a mainframe environment.

There is, at present, a notable lack of methodologies, abstractions and architectures to help programmers who are building distributed applications. It is one of the goals of this work to develop understanding of what forms these may take.

2.1.3 Relationship with Parallel Systems

At this stage we need to discriminate between distributed and parallel systems, as both research areas deal with communicating processes. Research on distribution and parallelism should be separated because they make different assumptions and have different goals.

- *Parallel systems*—Assume a single machine. When a parallel system runs on a set of machines, it is not designed to cope with partial failures, and in particular network failures. The main goals for a parallel system are normally performance and scalability.

- *Distributed systems*—Assume inter-connected but independent processes. One of the goals of a distributed system is to continue its local service despite remote failures. Compared with parallel systems, components in a distributed system should be prepared for network unavailability and delays, other components not responding, machines crashing, and all other kinds of partial failures.

For example, GUM [THM+96] is a parallel implementation of the functional programming language Haskell. GUM is available for a symmetric multiprocessor SPARCserver but also for networks of SPARCs and DEC Alphas. GUM uses an asynchronous message passing mechanism for communication, packs and un-packs data like an RPC, but failures are never mentioned in the paper cited above. In fact, one of the authors has confirmed [Tri95] that the system has been designed for performance in an environment without partial failures, that is, all partial failures are promoted to global failures. Despite this major drawback, performance remains the main issue for their future work.

Performance can be achieved by a parallel system up to a certain point. Very large applications, even if not geographically distributed, largely exceed the ability of super-computers.

An example is "warehousing" — where frequent business transactions use one computer, and data is backed up to a warehouse computer for processor-intensive database queries and data mining — partly because the same computer cannot cope with both tasks simultaneously.

In some cases there are other motives why a distributed system may be necessary, even if a parallel system could fulfill the job. For example, another reason for warehousing is that it matches the authority and responsibility assignments within the organisation. Accountants are responsible for an auditable and correct operational record, delivered promptly. They prefer not to have the managers directly use their data, which can also be translated into another format to better suit the managers (e.g., from COBOL files to a relational database on which general SQL queries can be performed).

2.1.4 Classes of Distributed Application

Many real-world distributed applications deal with mission critical data and communications to access that data. These applications attempt to achieve high-performance and availability despite network latency and failures.

However, it has been recognized that some put more emphasis than others in specific aspects of distribution. Birman [Bir93b], for example, classifies distribution into two different classes.

1. *Communication and Control* — A class of distributed applications that operate something, e.g., an air-traffic control or electricity distribution control system. These applications give importance to *correct behaviour under continuous availability* and to achieve that they typically replicate most needed data. This class of application usually monitors an underlying communication system, and when a failure occurs it immediately reconfigures and resumes operating. For example, Isis [Bir93a] was designed to support this class of application.

2. *Database Management* — In these applications there is a large amount of data maintained by servers that are shared by a number of clients, e.g., a banking or a hospital information system. A database style distributed application gives much more importance to the data that it is responsible for maintaining than to availability. Although continuous access to the data may also be of great interest, *it is the data itself and its consistency that are the central issue.* When a failure occurs, a high priority is to ensure that data does not become permanently inconsistent.

This dissertation is concerned with database-style distributed applications. However, the classification above can be considered to be over-simplified. For the purposes of this dissertation, distributed applications are re-classified in four classes.

1. *Real-time* — Includes part of the communication and control class above, plus all

other database applications with constraints based on "hard real-time" (never miss a deadline) or "soft real-time" (deadline can be missed, but preferably not).

2. *Data-based* — Basically the database class above. It includes remote access to databases (the typical client-server application) but it also includes all applications that maintain large amounts of data and need to use this data all over the application.

3. *Event-driven* — Applications that are waiting for external events, such as a fire alarm or a door closing. When an event occurs, a sequence of operations is called to respond to that event.

4. *Process-driven* — In this class of applications the events are generated internally; that is, there is a well-defined sequence of actions to follow during the life-cycle of some entity (typically a document).

In practice, most distributed applications have elements from more than one of these classes. For example, a real-time control system is both an event-driven and process-driven application, and will also have real-time constraints. Workflow applications, in which a document goes through a number of pre-defined steps across the organization, can be considered process-driven applications. But, at the same time, workflow can also be considered an event-driven application, as some events may occur from outside the application (e.g., document creation and deletion).

We can now restate the research areas that concern this dissertation. Since real-time systems are excluded, *the dissertation is concerned with data-based, event-driven and process-driven classes of distributed applications.* Together, these are of wider relevance than suggested by the initial classes proposed by Birman.

2.2 Models of Distribution

The previous section introduced distribution, its advantages and disadvantages, the relationship with parallel systems and the database style of distribution with which this dissertation is concerned. This section presents the two extreme models for building this style of distributed application.

1. *One-world Model* — Section 2.2.1 introduces distributed systems that manage many aspects of distribution and help programmers enormously to build distributed applications.

2. *Federated Model* — Section 2.2.2 presents an alternative view of how to build distributed applications that use a communication mechanism to permit independent components of an application (or applications themselves) to exchange data.

2.2.1 One-world Distribution Model

The one-world model attempts to hide the underlying distribution thus giving the programmer the illusion of a non-distributed system (see figure 2.1). Using this model, programmers interact with a single conceptual system which fully manages distribution.

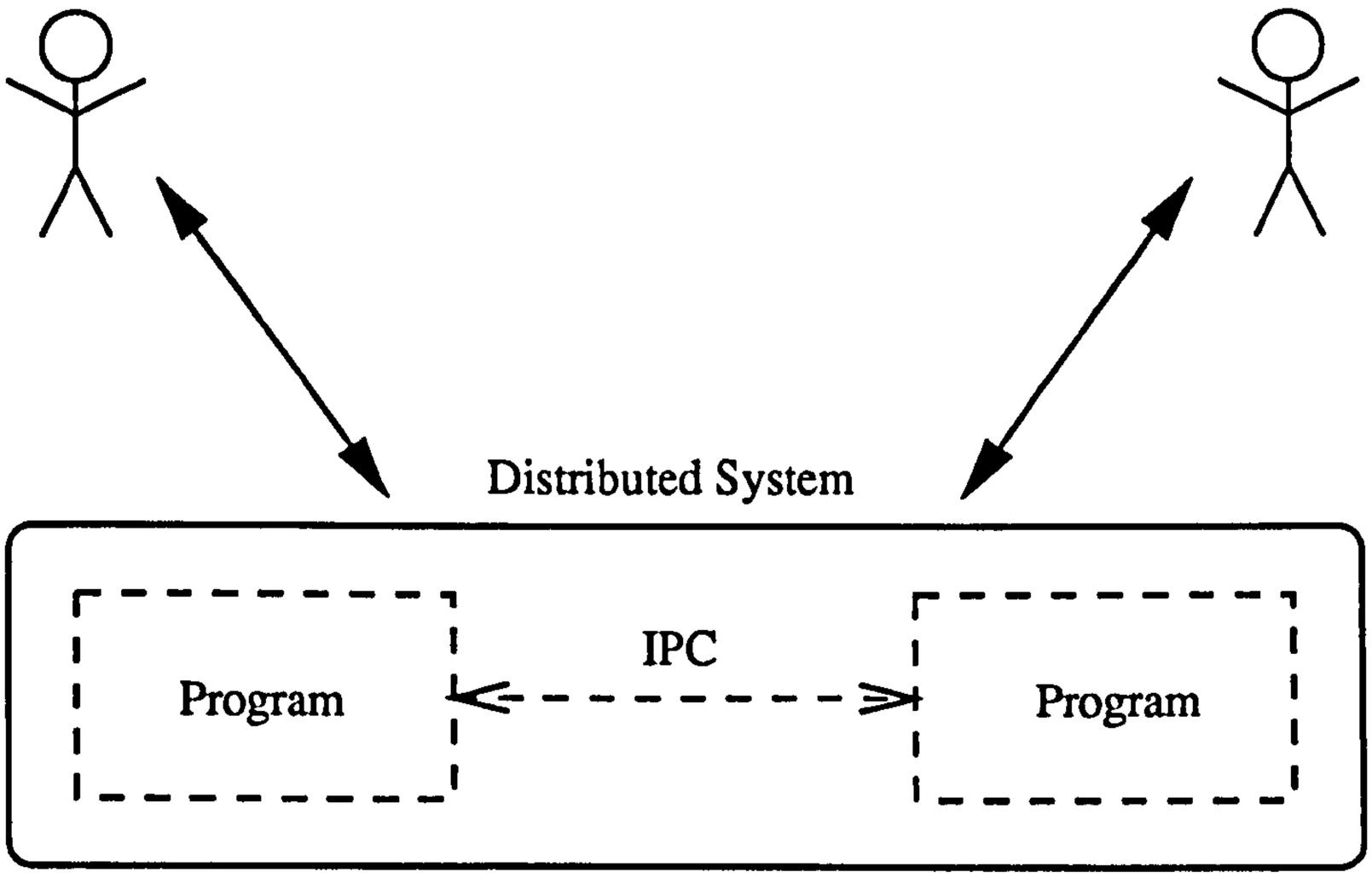

Figure 2.1: The one-world model of distribution

The one-world model of distribution may seem similar to a parallel system. However, a parallel system hides the multi-component nature of the system to achieve global performance. The objective of a distributed system, on the other hand, is to implement algorithms that enable the components to cooperate and to continue operating locally despite network and other partial failures. Performance is not the main objective in a distributed system, although it is an important one.

The one-world model is said to support distribution transparently. The ANSA reference manual [APM89] states that transparency exists when an undesirable characteristic of the distributed system is made invisible to the implementor of the application: "Transparency is the hiding of some aspect of the provision of a service from a user of that service". Conveniently, ANSA says that each aspect can either be *hidden* — when it is an "unnecessary or irritating complexity" — or *exposed* — when designers wish to exploit some service.

Seven transparencies are identified, each one hiding some aspect of a distributed service from the user.

1. *Access* — Hides the difference between local and remote provision of a service.
2. *Location* — Hides the location of the provider of a service.
3. *Migration* — Hides the effects of the provider of a service moving from one location to another.

4. *Identity*—Hides *from the provider of a service* the identity of the user invoking the operation.

5. *Replication*—Hides the difference between a replicated and a non-replicated provider of a service.

6. *Concurrency*—Hides the existence of concurrent users of a service.

7. *Fault*—Hides the effects of failures.

The tremendous advantage of this model is its simple conceptual framework that normally translates to a simple programming environment. The programmer does not need to understand the complexity necessary to manage distribution, deal with partial failures, optimise the placement of objects, or locate computations. In the one-world model all of these are performed automatically—and transparently—by the support system.

The one-world model is supported by distributed programming systems which can then be used to build distributed applications. These include distributed languages like Hermes [SBG+91] and distributed *object-based* languages like Emerald [BHJ+87], Argus [LS83, Lis84, Lis88] and Thor [LDS92, LAC+96]. The interested reader is referred to a survey including other research systems [CC91].

Attempts to build industrial distributed systems—as opposed to research prototypes—offering this model have to deal with all partial failures, network bandwidth and latency, communication costs, heterogeneity, performance and scalability, and still offer some degree of autonomy and availability. Because the one-world model tries to solve too many problems under the same umbrella, the result is a difficult compromise with a number of limitations.

1. *Scalability*—In order to support the illusion of being just a single world, many dependencies need to be created between nodes. These dependencies (e.g., remote references) are acceptable up to a certain scale, but they may prevent expansion of the system to a level where autonomy is needed for reasons of performance or physical, management and human organisation issues.

2. *Availability*—Also as a consequence of numerous dependencies between nodes, programs can progress only if the network, computers and programs are highly reliable because these programs then need many items of remote data to complete their computations. Such reliability is very expensive and difficult to achieve in any extensive environment.

3. *Autonomy*—Both reliability and dependencies between programs decrease autonomy for modifying the application. This includes changing parts of an existing application, adding completely new parts, and integrating the application with other applications.

Despite all these difficulties, distributed programming systems with support for databases offering the one-world model are now commercially available, e.g., ObjectStore [LLOW91].

Often they are based on the special case of a single server and multiple clients connected by a local area network. These will not scale easily to applications where multiple servers are required, or where the distribution of tasks does not map well to the client-server model.

The one-world model has its niche supporting highly specialised applications in local area networks connecting powerful workstations. Some engineering applications, such as CAD and CASE, fall into this category. However, the one-world model is inappropriate for large distributed applications as it assumes levels of reliability difficult to achieve with a reasonable cost in the real-world.

In the rest of the thesis we deliberately ignore distributed systems based on the one-world model of distribution. Namely, we will not discuss distributed shared memory, operating systems, file systems or relational database systems, which are the subject of general books on distributed systems [Mul93, CDK94]. Modern distributed systems based on the one-world model can also be found in surveys on distributed object-based programming systems [CC91] and distributed object-oriented database systems [Loo93, Che93].

2.2.2 Federated Distribution Model

Instead of attempting to hide distribution from programmers, the federated model of distribution offers to the programmer convenient tools — inter-process communication (IPC) mechanisms — to exchange data between programs (see figure 2.2).

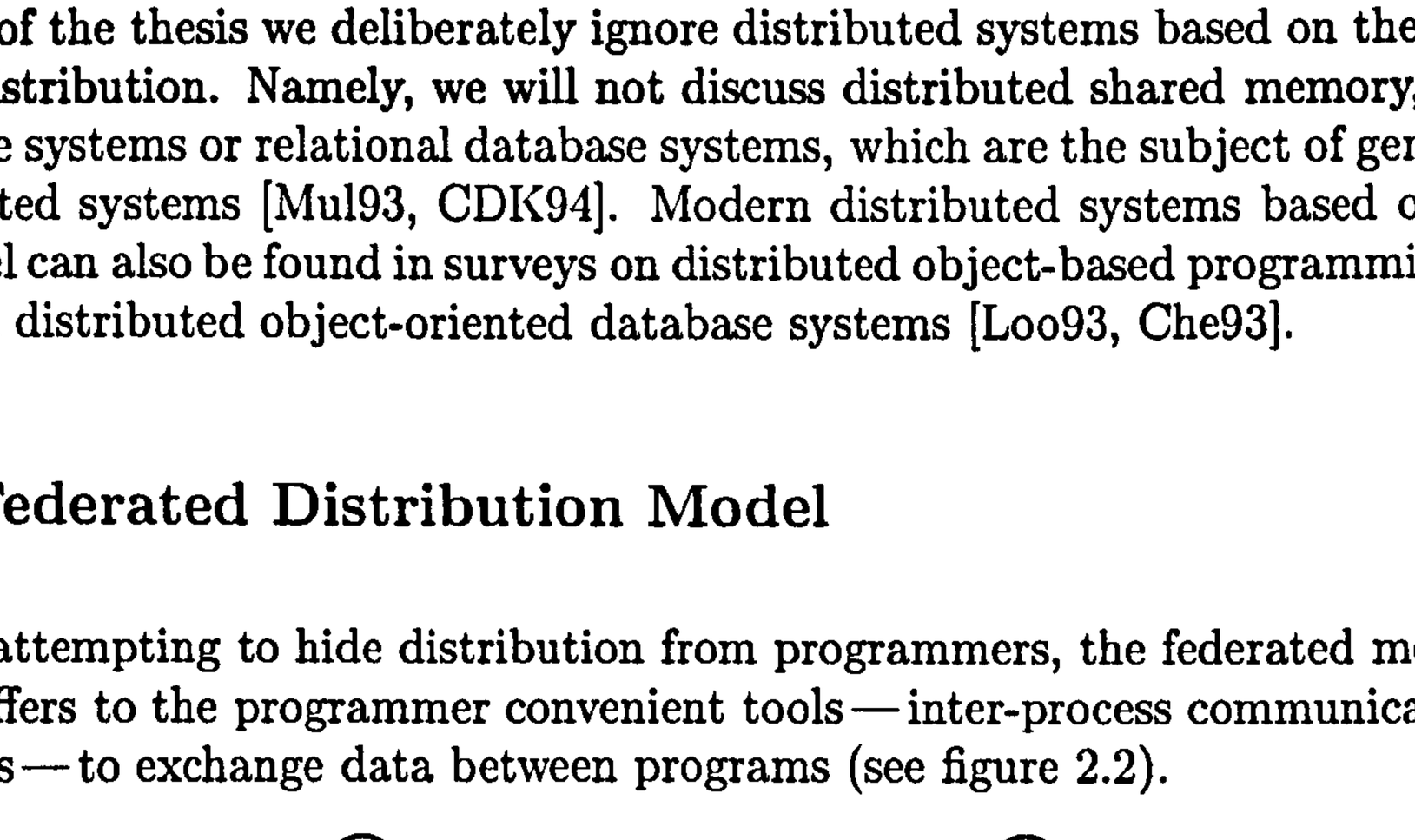

Figure 2.2: The federated model of distribution

The case for the federated model with an IPC mechanism is based on three assumptions.

1. As the scale of the application grows it becomes less feasible to hide the distribution and it may even be undesirable to hide it. For example, not only programmers but end-users themselves may be interested in the location of a failure.

2. Many applications will be built which use programs and data executing on different machines, although not all application programmers will have specialised knowledge of distribution. Using the federated model, large parts of the distributed application can be built as if they were local and then inter-connected by an IPC mechanism. Some applications already exist and need to be integrated with others (e.g., an accounting information system is already implemented in most hospitals.)

3. It is possible to assist application programmers in the task of writing such federated applications by providing better, yet simple and realistic IPC mechanisms. Given this assistance, application programmers are expected to be capable of building and maintaining sophisticated applications and of using the exposed properties of distribution.

As an extreme alternative to the one-world model one could envisage a distributed model that offers little help to application programmers, e.g., by exchanging byte streams using socket connections [Sun93c]. It is generally agreed that these leave application programmers facing too many difficulties. Instead, there are a range of positions which federated systems may take, and the federated models being built today are a compromise between these two extremes. One of the objectives of this dissertation is to identify a compromise between automation and referral to application programs.

The federated model has a number of significant advantages over the one-world model.

1. When the number of inter-dependencies between processes (programs being executed) can be restricted, the model is appropriate for *integrating autonomous systems* and thus access to legacy applications.

2. By supporting autonomy, the federated model *provides better support for scalability* and allows for local incremental change.

3. Federated models may also *support heterogeneity* by arranging that dependencies between programs are at a higher semantic level.

However, the federated model requires application programmers to deal with more aspects of distribution explicitly and thus requires them to master a more complex computational model. For example, they may now be responsible for managing the access to objects shared by several programs. In order to support reliable and highly available sharing, they may decide to replicate these objects across several stores which may create potentially inconsistent versions. Protocols for maintaining strict consistency between all replicas simply re-introduce the one-world model and are undesirable. But an application programmer may be able to exploit knowledge of the application to achieve consistency more economically just when it is needed. Compromises have to be found.

2.2.3 Motivation for Inter-process Communication

In general, the one-world model is worthwhile for distributed collections of programs up to the limit of current feasibility, a LAN for example. Where a part of the application which evolves as a unit and is managed as a unit can fit within that limit, it may be sensible to program it using the one-world model and tolerate it behaving as a single failure unit. This would depend on the relative cost of programming and service interruptions, and on the *mean time between failures* for the composite system.

Nevertheless, continuance of operation during partial failures, autonomy of evolution and management, and geographic distribution are always required by some larger applications. In these circumstances, the federated model with an IPC mechanism directly accessible to application programmers comes into its own, whether it inter-connects individual programs or entire applications built with the one-world model.

However, IPC cannot be restricted to its traditional role of just transferring data between parts of the application. Application programmers will expect to use the same set of rich features found in higher-order orthogonal persistent programming languages (see section 3.2.1) to help them build distributed persistent applications. This means IPC must also help programmers with some kind of system-wide naming scheme to maintain global referential integrity and consistency of object replicas. This IPC should also support modern concepts of information hiding, network agents, dynamic binding, and tolerance of heterogeneous environments.

2.3 IPC Design Issues

This section describes general IPC issues, for which a taxonomy is depicted in figure 2.3. The figure shows that from a set of high-level goals for any IPC mechanism, a number of basic design issues will arise. These design issues are also influenced by the restrictions at the implementation level. The figure also shows that two categories can be identified: an upper-level one concerned with semantics that is the subject of this thesis (above the dashed line in the figure) and a lower-level one concerned with more operational issues (below the dashed line).

The figure only gives an indication of the diversity of levels and issues involved, and it is not intended to be complete. For example, type-safety requires some kind of type-checking which is not represented in the figure. The four goals enumerated could be easily extended, e.g., with performance. IPC requires many other implementation details that are not described here. The levels themselves cannot be clearly separated: should performance be classified as a goal by itself, a design issue or an implementation detail ? Probably it is present in all three levels.

The IPC design issues that will be analysed in this section are in bold in figure 2.3. More specific design and implementation issues, such as the choice of a representation format for

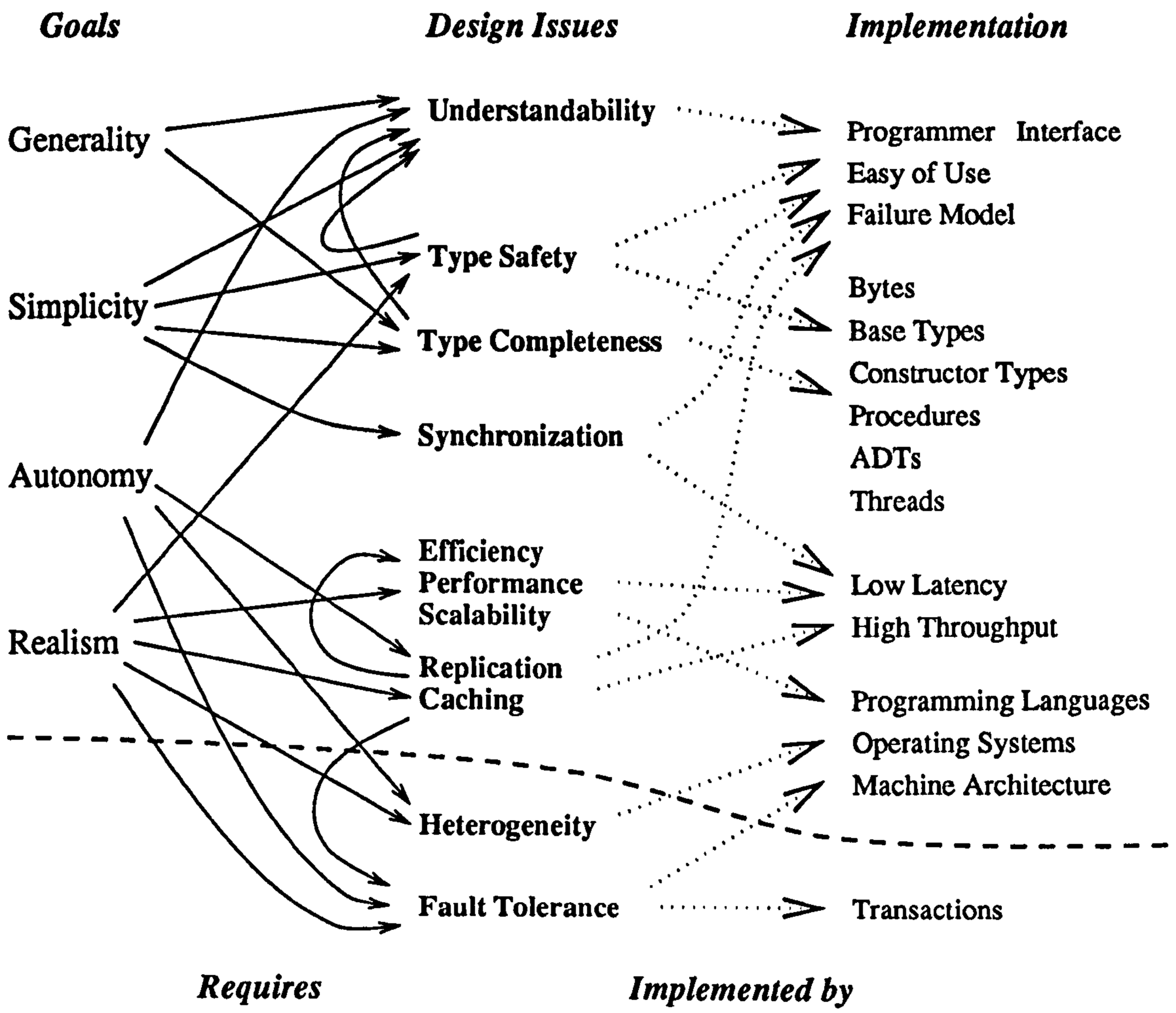

Figure 2.3: Taxonomy of IPC design issues

data transmission, will be described later in the context of RPC (see chapter 3).

2.3.1 Understandability

Distributed applications built using the federated model tend to be complex, a consequence of heterogeneity and partial failures that exist naturally in distributed systems. One of the single most important design issues for an IPC is for it to be *sufficiently simple* that application programmers can understand it and exploit its functionality in the intended way.

Simplicity was one of the primary motivations for RPC—extending the simple semantic model of the procedure call to a distributed environment. However, most well-known RPC mechanisms [BN84, OSF91, Sun93b] and even modern ones based on CORBA [OMG95] rely on an "interface description language" (IDL) to describe the remote procedures and their arguments, e.g., Java IDL [Sun96b]. (Sun has another RPC version for Java which does *not* use IDL [Sun96a].) This is useful—it separates the interface from the implemen-

tation—but forces programmers to learn an additional language, use a separate compiler to generate the RPC stubs, compile the stubs and link the compiled code to the main application.

As a result, some RPC systems have been designed with overall simplicity in mind. For example, Network Objects [BNOW93] concentrates on those "valuable features" of RPC—marshalling, type-checking, and efficient streams—and omits those considered "advanced"—transactions, heterogeneity, and intra-machine performance optimisation. The implementation is organised around a small number of simple interfaces and is compatible with high performance provided no failures occur.

2.3.2 Type-safety

A type system gives programmers a way to organise values in a programming language as sets belonging to types. Each type has a *representation* for its values and a *set of operations* permitted to manipulate the values of that type. For example, integers can be added and strings can be concatenated, but not vice-versa. A *type-safe language* prevents the application of any invalid operation for a type on values belonging to that type.

It is desirable to detect operations that infringe the type rules as early as possible (compile, binding or run-time) for three reasons [CW85, Con91].

1. *Safety*—If certain invalid operations will never occur at run-time, then programmers need not worry about type failures. This means no valuable programmer time has to be spent writing code to deal with type failures, and, more significantly, hunting for bugs caused by these failures.

2. *Protection*—Type-safety can also be used to protect the data in the store against accidental or deliberate abuse by using information hiding and viewing mechanisms [MBC+90].

3. *Efficiency*—If all invalid operations are detected at compile-time, then no type-checking is needed during execution.

In a local environment, "as early as possible" usually means at compilation time. However, compilation often depends on information provided by the programmer and this may become inconsistent: either programmers may not remember all changes and may make mistakes, or more commonly because the application is being developed by a team of programmers.

Dynamic (run-time) binding is the last opportunity to check for type violations before the program starts using the value. This has been recognized for many years in the persistent world because there are many cases in which static binding is not suitable [AM86, ABM88]. Persistent languages like Napier88 (see section 3.2) and modern languages like Java [AG96] support type-safe dynamic binding for this reason.

On the other hand, in a distributed environment each program or application sub-part may be independently written, compiled, linked and executed during construction or autonomous evolution. Without additional mechanisms, programs that communicate with other programs cannot be type-checked even at execution time. This means the efficiency, protection and safety provided by type-checking is potentially lost in operations involving IPC.

However, this loss of type-safety is especially important in a distributed environment because—as programs are now maintained not only by different programmers but also by different teams of programmers—the probability that invalid types will occur increases.

The importance of type-safety in the context of RPC has been recognized before. For example, type-safety is the first design issue mentioned by Hamilton for the Mayflower RPC [Ham84]. The compiler of the interfaces for the remote procedures generates a 62 bit value (UID) for every remote procedure that is guaranteed to be unique. Every remote call then sends this UID alongside the arguments to the remote procedure, and if the UID is not found locally the remote call fails with a "hard error" exception. Only client and server stubs generated by the *same* compilation can be used together and so they are guaranteed to be compatible.

Type-safety is even more important in a persistent environment [Con91]. While a type error in a conventional (non-persistent) program may corrupt the process address space—and the program can always be aborted and restarted—in a persistent programming language the "address space" is the entire store—and an error can therefore destroy long-lived, valuable data. The reader is referred to section 3.3.4 for a more detailed discussion of type-safety in the context of a persistent RPC. An example of the problems and the solutions adopted for our own type-safe persistent RPC are presented in chapter 4.

2.3.3 Type-completeness

Ideally, application programmers would like to use IPC with any data type supported by the programming language. Forcing application programmers to transmit bytes at the application-level—like MQseries [IBM94, IBM95] requires—is clearly not a desirable solution.

1. It loses the description and abstraction provided by types. It is helpful for programmers if the IPC can retain the same conceptual model across the interface.
2. It is complex and tiresome for programmers, who have to write procedures to pack and un-pack arguments and results themselves.
3. It is not safe, as all programmers will eventually make a mistake.

This is the reason why modern IPC mechanisms support at least most base types (e.g., integer, real and string) and a few simple constructors over these types (e.g., record and array).

Many languages, however, support a much richer set of data types. Between different languages, only the common set of types can be supported. But between programs written in the same language, programmers would like to exploit transmission of high-level values with more sophisticated types. Many distributed applications are built from programs written in a small number of programming languages (typically one, maybe two) so communication between them will often involve the same language at both ends.

If the IPC mechanism only supports a small sub-set of these, building a distributed application will require changes to existing local programming techniques. Worse, it forces programmers to convert local values to values suitable for transmission by the IPC mechanism, and then back at the target program. Even if the programming system helps with this—such as the pickling facilities in Modula-3 [BNOW93] and other languages [HL82, BJW87, Cra93, WRW96]—the programmer still has to be aware of, design for and make the conversion.

Another problem arises with a lack of type-completeness. As programmers get used to modern programming styles—such as using first-class procedures [AM85], objects and abstract data types (see next paragraph)—they will consider and use these types as they would use strings and structures in poorer programming languages. Also, as more and more languages support these richer data types, there is a case for augmenting the set of common data types supported by heterogeneous IPC systems.

For example, in Napier88 [MBCD89, MBC+94] procedures and threads are both first-class citizens. Both can be assigned to variables, passed as arguments to procedures and returned as results, or made persistent. A Napier88 programmer should be able to pass these advanced data types to other programs using an IPC mechanism. Some failure model will be required to deal with cases when the target program does not support these types.

Rich type systems are not restricted to persistent languages. In Java [AG96] and Modula-3 [Har92] objects are first-class values and contain both values and methods. It is natural for programmers to use these objects for IPC in the same way as they do within a process. This is the motivation behind Java's RMI [WRW96, RWW96] that can migrate an object of any type between processes (except objects bound to the underlying virtual machine, like threads [Eva96]).

2.3.4 Synchronisation

A concurrent programming language can execute several threads—also called light-weight processes—in parallel in the same address space. In order to maintain the integrity or consistency of the address space, some synchronisation mechanism is needed. Synchronisation is a fundamental issue also in distributed applications because there are necessarily several processes being executed concurrently.

A traditional answer to synchronisation is semaphores and monitors. Dijkstra [Dij68] first proposed semaphores as a synchronisation mechanism for Algol-60. Two operations, *wait*

and *signal*, control the access to some shared resource (e.g., represented by a variable). Requests for semaphores may lead to deadlocks. Also, when they are called is dependent on programmers and so their appropriate use cannot be enforced (thus mistakes can and do occur).

Monitors were first outlined by Dijkstra himself, then proposed by Brinch Hansen [Han73] and later refined by Hoare [Hoa74] to address the specific needs of concurrent applications. A monitor protects a resource and only one process can use the resource at any time. Java [AG96] bases its synchronisation on monitors.

An IPC mechanism can be presented as a shared queue that two (or more) programs concurrently try to access and manipulate. When one program wants to send a message to another program, it puts the message into the queue. Later, another program can pick up the message. This is called *asynchronous communication* because the source and target do not need to synchronise to access the queue.

Most IPC models, on the other hand, offer *synchronous communication*. When the source tries to send a message to the target, it will block and wait if the target is not waiting for a message. An IPC can be even more restrictive and support waiting for a certain message type only, e.g. Ada *rendez-vous* [Coh96].

While synchronous communication provides a simpler IPC semantics that is closer to conventional (i.e., sequential) programming, asynchronous communication has the advantage that it can exploit the inherent parallelism of distributed processes. (See section 3.3.10 for further discussion on asynchronous RPC.)

2.3.5 Efficiency, Performance and Scalability

These concepts are closely related and, as a result, sometimes confused. For the purposes of this thesis we define *performance* as the absolute wall-clock time required to execute an action. Poor performance then indicates that the IPC mechanism takes too much time when compared with the expectations of application programmers or related mechanisms.

Efficiency is defined as the inverse proportion of the amount of work performed to execute an action; if the system is inefficient then it needs more work to achieve the same result. An IPC mechanism can be efficient and not yield high performance, but it cannot have arbitrarily high performance without being efficient (assuming limited resources).

Scalability means the ability to retain acceptable performance as the complexity of the action to be executed increases. It is usually agreed that a system scales well when its performance decreases no worse than *linearly* in relation to the complexity of the action. An IPC system with excellent performance does not necessarily scale well if it relies on assumptions related to the number or small size of some components.

Efficiency, performance and scalability are usually measured in terms of clients per server, transfers per second, or amount of transferred data. The major approaches for improving

IPC performance and sometimes to achieve significant scalability can be exemplified as follows.

1. *Asynchronous IPC*—In many cases there is no need for an immediate response, not even a confirmation that the message has arrived. Message passing mechanisms and asynchronous RPC can achieve very high throughput and scalability by buffering IPC calls and sending messages in batches. For example, commercial systems—such as IBM's MQSeries [IBM94, IBM95]—support thousands of clients sending messages to a server by using a transactional queue. Other asynchronous systems tuned for control and communication such as Isis [Bir93a] also guarantee message delivery (provided there are no permanent failures).

2. *Same-machine Optimisation*—Only a small number of IPC transfers are truly remote, typically less than 10% [BALL89]. When IPC is used in the same machine, there are a number of optimisations which can be made. An improvement by a factor of 20 has been recently demonstrated by carefully designing the micro-kernel itself for IPC [Lie93].

3. *Operating System Optimisation*—Operating systems like Unix do not scale for thousands of processes operating on tens of thousands of files. Some IPC systems bypass the operating system and re-implement many of the operating system facilities themselves [Atk92b]. For example, relational database systems and transaction monitors support hundreds of (a limited class of short-term) transactions per second by buffering commit requests and processing them in batches.

Many IPC systems are designed for performance and scalability, and some examples were given above. But as we said already, performance and scalability do not necessarily coexist. For example, Lotus Notes [Lot96] supports thousands of clients accessing the same document database by replicating documents to the clients space, but its asynchronous mail-based data updates can hardly be thought of as offering high performance. Scalability in Notes is achieved by relaxing the consistency of documents, thus avoiding frequent (and costly) remote updates.

2.3.6 Replication and Caching

Data redundancy or replication exists when there is more than one copy of the same data. Replication is based on the assumption that for many objects the number of reads is much larger than the number of updates. By maintaining local copies of remote data likely to be read, replication improves performance and increases autonomy and availability. In a distributed object-oriented system this means two or more values for the same object, with different (local) identities.

Replication is used in the Andrew File System (AFS) for disconnected operation and scalability [HKM+88]. AFS claims it can support thousands of workstations using the same distributed file system. Client/server object databases cache *pages* at the client for

performance, e.g., ObjectStore [LLOW91]. More recently, elaborate models of caching based on *objects* have been used in Thor, a research-based distributed object database [AGLM95, LAC+96].

Many commercial systems also make use of replication. For example, high-end relational database management systems such as Oracle 7 also include basic support for replication (typically used for implementing data warehousing, see section 2.1.3). Groupware products such as Lotus Notes use replication for availability (e.g., disconnected operation) and for performance (reading and writing documents on the local disk instead of accessing a centralized document database across the network).

Replication can be even more explicit. For example, stashing [Bir88, ABC90] does not try to automatically keep the latest accessed data, nor does it try to maintain consistency between the original data and the local copies. Instead, it is the application programmer who is responsible for explicitly choosing the data to be stashed locally and for restoring consistency between the original and the local copy. Stashing only provides primitives to help organise and perform these operations.

Coherency Protocols

Replication is different from simply duplicating data because replication assumes a *replica coherency protocol* which guarantees that every copy has the same value as the original. A popular algorithm to achieve replica coherency is based on the idea of a *primary copy*. Using this scheme, one replica is chosen as the main replica; all updates to any other replica must first change the value of the primary replica.

The hope is that most updates will be made to the primary replica itself. However, some other mechanism has to guarantee that all other replicas have the same value as the primary copy. A trade-off has to be made between 1) updating all copies when the primary copy is updated and 2) checking the primary copy every time any other replica is used. In any case the primary copy scheme introduces dependencies between stores, latency times and communication costs for using replicas that can easily outweigh the advantages of replication.

Fortunately, *some applications tolerate inconsistencies* between copies up to a certain level. Also, for many objects the end-user or application programmer have enough semantic knowledge to decide which replica can, or is likely to, be changed.

In order to reduce the dependencies and costs associated with a strict consistency protocol, some replication systems relax the absolute guarantee for coherency stated above and instead only attempt to constrain inconsistency above a certain minimum. This minimum guarantee can be formally specified, e.g., a maximum period of inconsistency and that a local copy never makes a backward transition.

The big advantage of non-strict coherency is that it creates the potential for *batch synchronisation*, i.e., propagating together changes made to many replicas. This reduces latency,

communication and CPU costs, and permits these costs to shift to a time when they are less inconvenient.

For example, the distributed information system Hyper-G tries to maintain consistency cheaply by propagating changes using a probabilistic flood algorithm [Kap95]. Lotus Notes can synchronise a notebook to the centralized document database by e-mail. Data warehouses are typically loaded in the very early hours of the morning, when the operational database is not being heavily used.

But a non-strict consistency replication scheme has a major drawback: more than one replica can be changed between synchronisations. If this happens, at synchronisation time some *reconciliation* must occur to choose between the new values. Reconciliation is difficult because it often requires sophisticated semantic knowledge, e.g., human intervention. This is the reason why many replication systems prefer not to solve the problem at all; for example, Lotus Notes simply adds a version number to the filename if more than one replica has changed. Version numbers are useful if conflicts are rare, but become intractable if they become frequent.

Caching

Many distributed systems cache data to avoid a remote fetch if a local copy is available. Caching is a particular kind of replication based on the *primary copy* approach for replica coherency. Replication in general usually makes replicas visible, e.g., it is the system administrator who chooses which files to replicate in AFS (see section 2.3.6) and it is the end-user who decides when to synchronise the replicated documents in Lotus Notes. Although there is no clear border between caching and replication, caching typically attempts to provide transparent replication.

Commercial Web browsers automatically cache documents to avoid downloading the page again if it has already been fetched. For example, users of Netscape Navigator [Net96] can set-up the size of the memory cache (that restarts every time Navigator is called) and the disk cache (that survives Navigator executions). They can also choose whether the original documents are verified against the local copy "once per session", "every time" or "never". This example and Notes suggest even end-users can understand the complexities of distribution if provided with simple, well-defined primitives.

Change Propagation

A basis for all replication and caching protocols is how to detect, read, transfer, and write the differences between replicas, and especially *the level of granularity* to which the differences apply. For example, Lotus Notes checks if entire documents have changed and, if the replicas differ, the entire document is transferred.

Transferring entire documents when they usually only differ slightly can slow down the

reconciliation phase, especially if documents are large or the network has low bandwidth. On the other hand, detecting changes in small objects—e.g., parts of a document, even individual lines—may require CPU time that eliminates the advantage of reduced communication costs.

Another design issue for a replication protocol is *how to detect changes* to an object. For example, the *log* which keeps all updates to a relational database is an ideal source of changes because it is both small (compared with the size of the database) and is usually stored in a format that permits sequential scanning. Relational databases that support replication can simply send the log—or a relevant extract from the log—to update other databases.

Fingerprinting is another technique to detect changes. A *fingerprint* is a bit sequence that encodes a value probabilistically. In a *distributed make*, for example, fingerprinting can be used to detect whether some source code has changed since the last compilation [JV95]. Even though fingerprinting only gives a probabilistic answer, the risk of error can be made small enough for many applications. Similar mechanisms can be applied to replicated data.

Finally, another interesting issue is *where changes are detected.* Changes can be detected at the *mutation site* like in the relational databases and groupware products. These systems identify what has changed and propagate or *push* these changes to places where copies reside. In contrast, some systems like Web browsers delegate that responsibility to the *read site.* For example, the user can set-up Navigator to check if the original document is out-of-date (see *Caching* above).

Summary

This section 2.3.6 has introduced replication, caching and other variants. These are all based on protocols for maintaining the replicas consistency and techniques to detect which replicas have changed. This background will be used in chapter 6 to present our model of persistent spaces, in which a simple model of replication is used to permit some limited object sharing between persistent stores.

2.3.7 Heterogeneity

A heterogeneous system consists of diverse components. In a distributed system, however, heterogeneity can be revealed in many dimensions; not only is the system built from many sub-systems, but each sub-system itself may have different machine architectures, operating systems, programming languages, network technologies and protocols, and database systems. Each of these components can potentially use a different convention for representing data values.

Support for heterogeneity—also called "openness" in marketing parlance—is especially important in these days of client-server, company mergers and acquisitions, and connectiv-

ity between existing information systems. For example, the commercial message passing mechanism MQSeries [IBM94, IBM95] runs on 11 operating systems and can be used from 4 languages, all with a similar API. The relational database Oracle runs in more than 60 combinations of operating systems and machine architectures, while supporting virtually the same SQL interface. This extensive support for heterogeneity is considered to be one of their major selling points.

Despite different programming languages, operating systems and machine architectures, ultimately all data is transmitted between programs as bit streams. Using the same agreed data format and transport protocol, any two programs can communicate with each other. (See section 3.3.7 for a discussion of transport protocols and sections 3.3.8 and 3.3.11 for more on data formats).

However, being able to exchange data structures is a limited form of heterogeneity: different programming languages may support different data types or even have different semantics for the same types. There are two traditional solutions to this problem.

1. *Restrict the data types supported*—Often the available protocols operate only to preserve low-level types, and fail to help programmers interchange data at the conceptual level at which they are working. Examples include most RPC systems (see section 3.3).

2. *Restrict the range of applications*—Another approach has been the development of agreed type systems with which to interchange well-defined, application specific data. Examples of such specific high-level protocols are EDI [Pre96], HTML [Wor96] and OLE DB [Bla96b, Bla96a].

It is the combination of these issues described above that make heterogeneity so difficult: 1) rich, high-level types; 2) not specific to any application and 3) language independent. In this thesis we will ignore language independence in order to concentrate on support for general purpose mechanisms for high-level types such as procedures.

2.3.8 Fault-tolerance

A system which is tolerant to failures should have been designed and implemented accepting that these are an unavoidable consequence of operating in the real-world. These systems have a *failure model* that explains what happens when a failure occurs. Typically, the simplest failures are dealt with automatically, whereas unrecoverable failures force the system to slow down and eventually to stop with the minimum of undesirable consequences (such as destroying long-term valuable data). Fault-tolerant systems also help applications to restart—also called "recovery"—after the failure has been repaired.

Many fault-tolerant systems are based on transactions. A transaction is an atomic action that either *succeeds* (so the changes made during the transaction cannot be undone later)

or it *aborts* (so it is as if the transaction never occurred). Transactions are useful for implementing fault-tolerance because they maintain data integrity in the event of failures and help recovery later. They also present a simple semantics for the application programmer (and eventually to the end-user).

In addition to failures, distributed systems also have to deal with *partial failures* in which only part of the system fails. Partial failures are characterised by three properties that distinguish them from global, "stop the world" failures [WWWK94].

1. Partial failures are orthogonal to program execution.
2. No single part of the system is able to determine which component has failed.
3. No global state exists for recovery.

These properties create much additional complexity because they introduce "indeterminacy", that is, there is no easy way to discover if a remote computation was completed successfully, partially completed or not even initiated. Reducing the number of partial failures is always feasible—for example, using replicated communication channels, more reliable computers, tested software—but it becomes increasingly expensive in capital and operational costs.

One approach to solving the "indeterminacy" problem is to provide in a distributed system the simple semantics of transactions: all or nothing, and recovery to a stable state. For example, the *two-phase commit algorithm* [CDK94] guarantees that for changes made on many databases either all commit or none commit. Distributed transactions have a role in any distributed system, but do not provide all the coordination if there are processes that depend on data outside a participating database (e.g., in the file system). Moreover, these algorithms may not scale or support heterogeneity.

This is the reason why, instead of global distributed transactions, some IPC mechanisms offer transactions as a feature of the communication itself. For example, products that implement transactional queueing, such as IBM's MQSeries [IBM94, IBM95] and Encina RQS [Tra93b], guarantee that a set of messages put in the queue by a client will either all arrive at the server or none will. This is potentially much easier (and cheaper) to achieve than global transactions, and still of great help to application programmers writing debit-credit operations, for example against a banking computing system.

A related use of transactions in the context of IPC can be exemplified by a transaction monitor, such as Transarc Encina [Tra93a], which acts as a back-end to the IPC mechanism at the server. Although not part of the communication itself, a transaction monitor can help to improve communication performance by accepting messages at a very high rate from a transactional IPC mechanism. These messages are processed only after communication has ended, probably also against a transactional database.

Although the research described in this thesis does not attempt to deal with failures directly apart from providing a simple failure model, the techniques presented in chapters 5 and

6 can be used to achieve similar fault-tolerant behaviour for the distributed application. These techniques include a separation between local and remote computation, replicated values, no hidden dependencies between stores, and communication time reduced to a minimum. The application programmer can then compose these primitives in a variety of ways to achieve a good compromise for tolerating partial failures in the distributed application.

2.4 Summary

This chapter has presented a motivation for federated distributed applications and how IPC can help to build these. We have discussed why the one-world model of distribution may be suitable for applications which behave like a unit, but it will not scale for the kind of large, long-lived distributed applications which we wish to support. A number of IPC design issues were then described: understandability; type-safety; type completeness; synchronisation; efficiency, performance and scalability; replication and caching; heterogeneity; and fault-tolerance.

The examples given in this section show that many IPC mechanisms have been designed and built for supporting specialised abstractions, e.g., the guaranteed delivery of messages based on a transactional model. Not much attention has been dedicated to migrating instances of any type, especially richer types like procedures and instances of abstract data types. The special needs of persistence have not traditionally been dealt with either. We conclude that it is still necessary to look for other solutions to facilitate building a certain class of large distributed persistent applications.

Chapter 3

Overview of Persistence and RPC

The research presented in this thesis attempts to combine RPC and persistent programming. In this chapter we first introduce persistent systems and Napier88, the persistent programming language we have chosen for our experiments. We then continue by describing the RPC design issues that are generally considered fundamental and present some existing solutions to these. We finally review the most important features of persistent systems and the new issues that arise from combining persistence with RPC.

3.1 Orthogonal Persistence

The *persistence of a data object* is the period of time for which the object exists and is usable [ABC+83]. A *persistent system* is a computational system that manipulates all of its data equitably irrespective of their life time.

For example, one can conceive a *persistent computer* that uses a combination of hard disks and RAM backed up by batteries to give the illusion of a single high-capacity permanent storage mechanism with very fast access. Another possible form of supporting persistence is by means of a *persistent operating system* that gives the same illusion using a conventional computer architecture. However, a particularly useful—and thus popular—form of persistent system is that offered at the programming language level.

A *persistent programming language* ensures that values remain available as long as they are required for computation, thus eliminating the need for files or databases. By contrast, a conventional programming language manipulates directly only values which are resident in memory. If these values are to be used later by the same or another program, then the data has to be explicitly transferred from (volatile) program data structures to some sort of permanent (non-volatile) storage utilising different manipulations, representations, naming schemes, and so on.

A commercial object-oriented database system like ObjectStore [LLOW91] supports some

level of persistence by extending an existing object-oriented language, typically C++. However, extending C++ to support persistence is a non-trivial task, especially if the goal is persistence that is orthogonal to the use of data. *Orthogonal persistence* [ABC+83, AM95] is only achieved by applying three principles.

- *Persistence independence*—The form of the program is independent of the longevity of the data that it manipulates, i.e., programs look the same whether they manipulate short or long term data.
- *Data type orthogonality*—All objects should be allowed the full range of persistence irrespective of their type, i.e., there are no special cases for certain types [ACC82].
- *Persistence identification*—The choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system, i.e., persistence is not restricted to a subset of types.

Many so-called persistent systems do not support orthogonal persistence. ObjectStore, for example, stores methods in operating system files while data are stored in a database. By separating the data from their operations, a persistent system like ObjectStore allows a programmer to delete methods of a class for which objects may still exist in the persistent store, although there is no problem to delete methods of a class with no persistent objects. Thus methods for persistent objects behave differently from methods for non-persistent objects, even though all methods are written in the same programming language.

The only way to achieve orthogonal persistence is to store the code in the database together with the data it manipulates.

3.1.1 Benefits of Persistence

The benefits of persistent systems, described extensively in the literature, have been summarised in a recent survey [AM95]. These are reproduced below.

- *Increased programmer productivity*—Orthogonal persistence frees application programmers from writing and maintaining code to move data between the programming language (e.g., C++) and the database system (e.g., Oracle). Perhaps even more importantly, programmers need not write code to translate the data between different representations (e.g., object-oriented and relational data models) because all data remains in a single system for as long as it exists and is usable.
- *Better data protection*—All data in a persistent system are constrained to the operations permitted by the type system, thus no invalid access to any data can occur. This means type-safety extends to long-term data [Con91]. Also, because data can only be accessed from the persistent language, access control and software constraints (e.g., procedural encapsulation) can be imposed on all data.

- *Referential integrity*—An object is made persistent if it is reachable, directly or indirectly, from one or more *persistent roots*. This way of identifying persistent objects guarantees referential integrity because all objects referred by any other persistent object will also be made persistent.

- *Incremental software construction*—Large persistent systems can be built incrementally by adding new parts to the schema, inserting new data and installing new programs.

In short, orthogonal persistent systems are expected to provide better support for the design, development, operation and maintenance of complex database applications than the traditional solution based on a programming language using a separate database system.

3.1.2 Implementation Strategies

Persistence seamlessly integrates a database system into the programming language itself, so at least a mechanism to execute programs and another to store data are required to implement a persistent system. This combination has been achieved in a number of ways, described below, by increasing commitment to the persistence philosophy [AM95].

1. *Providing a library of persistent facilities in a standard language*—The simplest method to support persistence. Examples include libraries to store data on file systems and access relational databases, e.g., Java's own version of ODBC [Sun96c].

2. *Extend an existing system with facilities for the other system*—This approach starts with an existing system, either a database or a programming language.

   - *Extend an existing database system with a more complete type system and computational facilities*—Examples include Postgres [SR86, RS87] and SQL3 extensions to SQL supported by commercial relational database systems [Mel96].
   - *Extend an existing programming language with database facilities*—Examples include database programming languages such as Pascal/R [Sch77], the first version of PS-algol [ABC+83], and the current generation of object-oriented database systems [LLOW91, Deu91].

3. *Design and implement a new persistent system from scratch*—Until now only a research approach, even though some of the resulting persistent systems have been used by industry on a limited scale.

   - *Persistent hardware architecture*—The computer itself provides persistence, so everything running on the computer is persistent. For example, the MONADS architecture [Ros90].
   - *Persistent operating system*—The operating system implements a persistent system on top of a conventional architecture. Examples include Grasshopper [DdBF+94] and EOS [Gru92, DG92].

- *Persistent programming language*—The programming language seamlessly integrates databases features. Fibonacci [ABGO95], Tycoon [MMM93, MMS94] and Napier88 (see next section) are typical examples.

Only those in the third category (designing a new system from scratch) have the potential to deliver the full benefits of orthogonal persistence [ABC+83, Atk92b, AM95]. Persistent programming languages are popular because they can run on conventional operating systems.

3.2 Napier88

Napier88 [MBCD89, MBC+94] is a procedural, orthogonal persistent programming language that also incorporates a number of other interesting features that are described in this section.

3.2.1 Features

The relevant features of Napier88—especially those directly related to this thesis—are described below.

Orthogonal Persistence

Persistence in Napier88 is orthogonal to the type of the object and defined by reachability from a single persistent root. The persistent root, returned by a procedure called `PS()`, is by convention an *environment* (see dynamic binding below) which contains names that bind to objects, including other environments. The named parts of a store can thus be organised in a way similar to a hierarchical file system. Since persistent recursive types are supported, the store contains an arbitrary graph of data.

Rich Type System

The Napier88 type system is described by a set of scalar types and a set of constructors, and the recursive composition of these [Con91]. Procedures are just a normal data type in Napier88 so they can be created at run-time (see *Reflection* below) and put in the store. The language also supports abstract data types [Cut93] and parametric types, including parametric procedures. The type of an object can be obtained at run-time, and type equivalence is structural.

First-class Procedures

Procedures in Napier88 can not only be declared, passed as parameters, and executed, but can also be assignable, the result of expressions or other procedures, elements of structures and vectors, and they can be made persistent. First-class procedures can be used to implement abstract data types, modules, separate compilation, views and data protection [AM85].

Dynamic Binding

While static binding and type-checking are desirable features in a programming language, dynamic binding is also needed in a persistent language where some types or even some object names are not known at compile-time [ABM88].

Dynamic binding in Napier88 is implemented with collections of bindings called *environments* [AM90, Dea89]. A *binding* is a tuple containing a name, a type, a value and information indicating whether the value is constant (i.e., constancy is not a property of the type). Because procedures are first-class values in the language, dynamic binding supports the incremental construction of programs.

Type-safe Linguistic Reflection

Reflection permits a program to modify or extend its own behaviour at run-time. In Napier88 this is achieved by allowing running programs access to the compiler, which is a standard procedure in the store; programs may alter themselves by creating new program fragments or even new types at run-time, which are compiled and integrated into the current execution [Kir93]. Type-safe reflection means that all reflective operations are type-checked.

Concurrency and Concurrency Control

Concurrency in Napier88 is achieved by threads of execution and critical regions [Mun93]. Threads are offered to the programmer as an abstract data type with operations to create a new thread, suspend a thread, kill a thread and so on. Critical regions are implemented with semaphores, first proposed by Dijkstra [Dij68].

Conclusion

Perhaps most important in Napier88 are not these features *per se* but the synergy that is created when they are used in conjunction. For example, type enquiry together with reflection, persistence and dynamic binding enables the programmer to create customised

programs at run-time and put these in the store to be re-used later (see chapter 4). The persistent workbench (see section 3.2.4) makes extensive use of many advanced Napier88 features, as well as many of the tools incorporated into the workbench itself.

3.2.2 Type System

Persistent systems, and Napier88 in particular, attempt to provide a rich data model for both data modelling and protection [MBC+90, Con91]. The Napier88 type system consists of a set of base and constructor types that will be important for explaining the three IPC models proposed in chapters 4 to 6.

Base Types

Values of a scalar type are immutable. When a value of one of these types is passed as an argument to a procedure, *a local copy is made*; updating this copy has no effect on the original value passed as an argument. Napier88 supports the following scalar types.

- *Integer*—The set of all integers.
- *Real*—The set of all reals.
- *Boolean*—With two values, true or false.
- *Null*—With one value, nil.
- *String*—A sequence of characters of arbitrary length.
- *Graphic types*—Pixel (to be used as elements of images, see below) and pictures (consisting of transformable line drawings in 2D real space).

Constructor Types

Values of constructor types are defined by the use of type constructors, and the recursive composition of these. When a value of one of these types is passed as an argument to a procedure, *only a reference to the original value is copied*; updating the values reachable from this reference changes the original values. Napier88 supports the following constructor types.

- *Vector*—One dimensional array (N-dimensional arrays can be formed by the recursive use of this constructor).
- *Structure*—Record of named and typed fields.

- *Variant* — Discriminated labelled union.
- *Image* — Rectangular array of pixels.
- *Procedure* — Function with or without result.
- *Abstract data type* — This is a structure for which the type is abstracted over [MP85, CDMB90]. Because the type is abstract, the fields of the structure are usually procedures to manipulate the abstract value.
- *Environment* — Variable set of bindings (see *Dynamic Binding* in section 3.2.1 above).

There is also the type *any*, the infinite union of all types.

In addition, the following constructors can be parameterised with type variables which must be consistently substituted in order to produce a usable type: vector, structure, variant, procedure, and abstract data type. This means a type can be defined generically and only instantiated to a concrete type when it is used.

Parametric procedures are a special case as they can be used even in an abstract form. For example, an identity procedure that returns its only argument does not need to know the argument's type. In order to emphasise this difference, parametric procedures in Napier88 are called *polymorphic procedures.*

3.2.3 Implementation

Napier88 was originally designed to be the successor of PS-algol [ABC+83] as part of the PISA project [AMP87]. The language was first implemented in 1987–1989 at the University of St. Andrews by Ron Morrison and his team, and has been evolving ever since.

The implementation of the most recent Napier88 Release 2.2 (1995) is usually described in terms of four main modules.

1. *Programming language* — As defined in the reference manual [MBC+94].
2. *Source code compiler* — That generates byte code. (The current version of the compiler is implemented in Napier88 itself [Cut93].)
3. *Abstract machine* — Needed to interpret the byte code [CBC+90a]. (There is also a new implementation called PamCase [CCM95] that will be in normal use very soon.)
4. *Stable store* — To reliably maintain data and programs [Mun93]. (The stable store can also be considered as part of the abstract machine.)

Napier88 is currently available for Sun SPARCs running SunOS 4.1 and DEC Alphas with OSF/1. (It also runs on Sun with Solaris 2 as a SunOS application.) At the University of Glasgow, Napier88 has been used for seven years in a large number of student and research projects. Napier88 is now being used extensively in about 50 locations around the world.

3.2.4 Programming Environment

Napier88 includes not only a programming language but also a complete programming environment—libraries, methodologies and tools—that complement the language and help programmers to build Napier88 applications. Key examples are given below.

- *Standard Library*—Described in the Standard Library Reference Manual [KBC+94], contains necessary procedures such as those for I/O, failure management, communications, graphical programming and in general an interface to the outside world.

- *Glasgow Libraries*—These complement the Standard Library with bulk types (lists, maps, and so on) [ABC+93], additions to WIN (see item below), the RPC described in this thesis, and other useful procedures and data. The libraries are well documented, including code examples [WWP+95].

- *WIN (Windows In Napier88)*—A graphical user interface (GUI) toolkit for providing graphical interfaces within Napier88 applications [CDKM89, Kir93].

- *Hyper-programming environment*—Based on WIN, the hyper-programming environment comprises a set of tools to locate data items in the persistent store and to display and edit hyper-programs [Kir93]. A *hyper-program* represents a Napier88 program in the store and is composed of source code and bindings from that source code to the objects needed by the program.

- *Programming methodology*—Persistent programmers can use a *methodology* for building, changing and extending programs in Napier88 [Sjø93]. These guidelines include the division of the application into modules—defined by the programmer and dependent on the application—and the development of each module as separate programs for initialising, loading and deleting procedures and data structures. There is a prototype to support this methodology [SWA+95].

- *Programming workbench*—The workbench is a programming development environment to build persistent applications [AKP+94, WPA+95, SWA+96]. The workbench includes tools to display, edit, group, compile and execute programs; to visualise the contents of the persistent store [Lav95b]; to create and maintain software libraries; and to find components of these libraries [Bro93].

In addition, many other general purpose tools have been developed over the years to help with Napier88 programming, in particular a persistent extensible command interpreter called **hcs** [WPA+95]. Because these commands are executed against a warm persistent cache, **hcs** supports efficient compilation and execution of Napier88 programs via a textual interface.

3.2.5 Limitations and Challenges

Orthogonal persistence is a simplifying concept that has a number of important benefits as described in section 3.1. However, these benefits cannot be allowed to obscure its current limitations, especially those shown by the particular implementation of Napier88 we have used for the experiments described in this thesis. These limitations are described here for completeness.

Orthogonal Persistence

The implementation of orthogonal persistent systems presents difficult challenges because they live within, and need to interact with, a non-persistent world. For example, the Napier88 `file` type is supposed to be persistent. However, real UNIX files that Napier88 opens *cannot* be totally controlled by the persistent system. This means that operating with objects of type `file` in Napier88 is different from operating with all other persistent objects. The programmer has to be aware of these differences. For example, sometimes the programmer has to use UNIX semantics such as file error codes.

Binding Complexity

The large number of binding mechanisms that are now available in Napier88 can be difficult for application programmers to understand. The degrees of freedom supported by a flexible binding mechanism — constancy or variability, L or R values, and four different binding times (composition, compilation, linking and execution) — potentially generate 16 different kinds of binding [AM88, ABM88, Kir93]. The case here is not whether they are useful or not, as they are, but how to explain these binding mechanisms so that programmers know when to use each of them.

Performance

The performance of the current Napier88 implementation — based on byte-code interpretation — is 1 to 3 orders of magnitude slower than C. (Performance measurements of Napier88 and their comparison with C will be presented later in chapter 7.) Even though persistence systems augment traditional programming languages with several features, there seems to be nothing fundamental precluding an efficient implementation of a persistent programming language.

Some of the problems may be due to historical reasons; others are due to the fact that Napier88 is a research language. Optimisations would impede some of the research experiments by making the compiler more complex to change. A forthcoming implementation called PamCase [CCM95] promises to ameliorate this problem, partly because the store size is significantly reduced.

Threads and Multi-user Support

Concurrency in Napier88 is supported by user-level, pre-emptive threads implemented by a "round-robin, fixed time-slice by the number of instructions" [Mun93]. The entire process stops when a thread blocks for I/O. Concurrency control is based on semaphores, which are now considered too difficult to use and error-prone due to their potential for deadlocks. Although threads and multiple "sessions" (a top-level window in a workstation) give some limited multi-user support, Napier88 does not support nested transactions.

Heterogeneity

Napier88 has limited support for interacting with other languages and systems. A challenge that has been recently discussed is to open Napier88 to the outside world in order to facilitate the test and eventual acceptance of persistence by other communities. A preliminary experiment integrating Napier88 with Tcl/Tk [Lar96] suggests this is both feasible and does not violate the persistence abstraction.

3.2.6 Comparison

Table 3.1 compares Napier88 with C++, Java [AG96] and a typical object-oriented database system (OODBMS) such as ObjectStore [LLOW91] *from a programming language point of view* (not its current implementation). We choose C++ because at the time of writing it still represents the industry standard object-oriented language and is a well-known language. Java is a potential, future, object-oriented language standard, as Java or a similar type-safe, well-defined language will eventually replace C++. Finally, the OODBMS represents persistence as it is manifested commercially today.

We now define some of the terms used in the table. By "Ease of use" we mean the *learning curve* for the language as well as its use by an average application programmer. "Product" means an implementation of the language is available commercially. The fact that Napier88 is the only one in the table which is *not* a product may explain some characteristics of Napier88 not listed here, such as its relatively poor performance.

A language is "object based" if it supports classes or abstract data types that encapsulate an object's state and support a method interface. "First-class procedures" are supported in the language if it treats procedures like any other data type. By "dynamic binding" we mean the ability to add new programs to the current execution *in a type-safe manner* (thus excluding C++ and the OODBMS). A language is "neutral/portable" if the source code is independent of the particular language implementation and the environment where the program is compiled or executed.

A simple classification based on "—/Yes" was chosen for clarity purposes, although most features would require a more elaborate classification and detailed explanation if Napier88

Programming Language Feature	*Programming Language*			
	Conventional		*Persistent*	
	C++	*Java*	*OODBS*	*Napier88*
Persistent	—	—	Yes	Yes
Ease of Use	—	Yes	—	Yes
Type-safe	—	Yes	—	Yes
Product	Yes	Yes	Yes	—
Object-based	Yes	Yes	Yes	Yes
Inheritance	Yes	Yes	Yes	—
First-class Procs	—	—	—	Yes
Garbage Collection	—	Yes	—	Yes
Dynamic Binding	—	Yes	—	Yes
Neutral/Portable	—	Yes	—	Yes
Threads	—	Yes	—	Yes
Exceptions	—	Yes	—	—

Table 3.1: Scorecard for Napier88 and related systems

was the main subject of this thesis. We now discuss the major disadvantages of Napier88 when compared with these related languages and systems.

Lack of Exceptions

Although the Standard Library promotes a model to deal with failures by replicating each environment with an "error environment" that simulates exceptions, the experience with Glasgow Libraries suggests this model is seldom used elsewhere. Instead, a more traditional approach based on procedure results that may also represent an error code is typically used, such as `variant` return types.

Lack of Inheritance

The lack of inheritance in the current implementation of Napier88 can be considered a potential disadvantage. Although inclusion polymorphism has been proposed for Napier88 [Con91], the integration between sub-typing inheritance and mutable values presented some problems [CMM91]; these are currently being investigated [CBM96].

Not a Product

Although the provision of orthogonal persistence for well-known languages is being discussed [CO94], under development [AJDS96, ADJ+96], and already exists in some cases [ODI96], Napier88 at present has no commercial implementation. This could mean limited technical support and not so many third party libraries and tools for Napier88. However,

both the University of St. Andrews and the University of Glasgow provide a support environment, extensive libraries and programming tools. On the other hand, not being a product gives freedom to evolve the language and thus provide its users with the latest available research technology at any one time.

3.3 Remote Procedure Call

In this section we will introduce RPC and describe the most important RPC design issues that are related to the research described in this thesis. There are some natural overlaps between the issues described here and those presented earlier in section 2.3 for IPC in general. However, in this chapter we concentrate on specific, lower-level design issues particular to RPC.

3.3.1 Introduction

A *remote procedure call* (RPC) is a paradigm for providing high-level communication between programs. Using an RPC mechanism, a procedure in one program of a distributed application may call a procedure in another (remote) program. (By "remote" we mean in a different address space, potentially on a different machine.)

It was Birrell and Nelson [BN84] who first made RPC popular and the PhD thesis of Nelson [Nel81] is considered to be the first work in the field. However, Staunstrup [Sta82] cites at least two earlier RPC mechanisms: the typical synchronous, blocking RPC found in *distributed processes* [Han72]; and an asynchronous RPC mechanism embedded in the programming language *Concurrent Pascal* [Han75].

Using RPC has a number of advantages over more traditional (lower level) communication paradigms like sockets [Sun93c].

1. *Clean and simple semantics*— RPC is based on procedure calls, a well-known mechanism for transfer of control and data in programming languages.

2. *Susceptible to type-checking*— Type-checking is not only possible but also natural because RPC offers an interface at the programming level.

3. *Potential efficiency*— Because data conversion performed by the RPC would have to be made by the programmer if other lower-level mechanisms were used.

An RPC mechanism is usually described in terms of two main modules: a *client*, the caller program; and a *server*, the callee program (see figure 3.1). The client normally executes on one machine and the server on another, but nothing in the mechanism prevents the client and the server from being executed on the same machine (even in the same address

space) nor the client from being a server to another client. For each side there is a further partition into three main layers: *user program*, where user procedures are implemented; *stub procedures*, for packing and un-packing values into and out of messages (see below); and *transport protocol*, for exchanging these messages across the network.

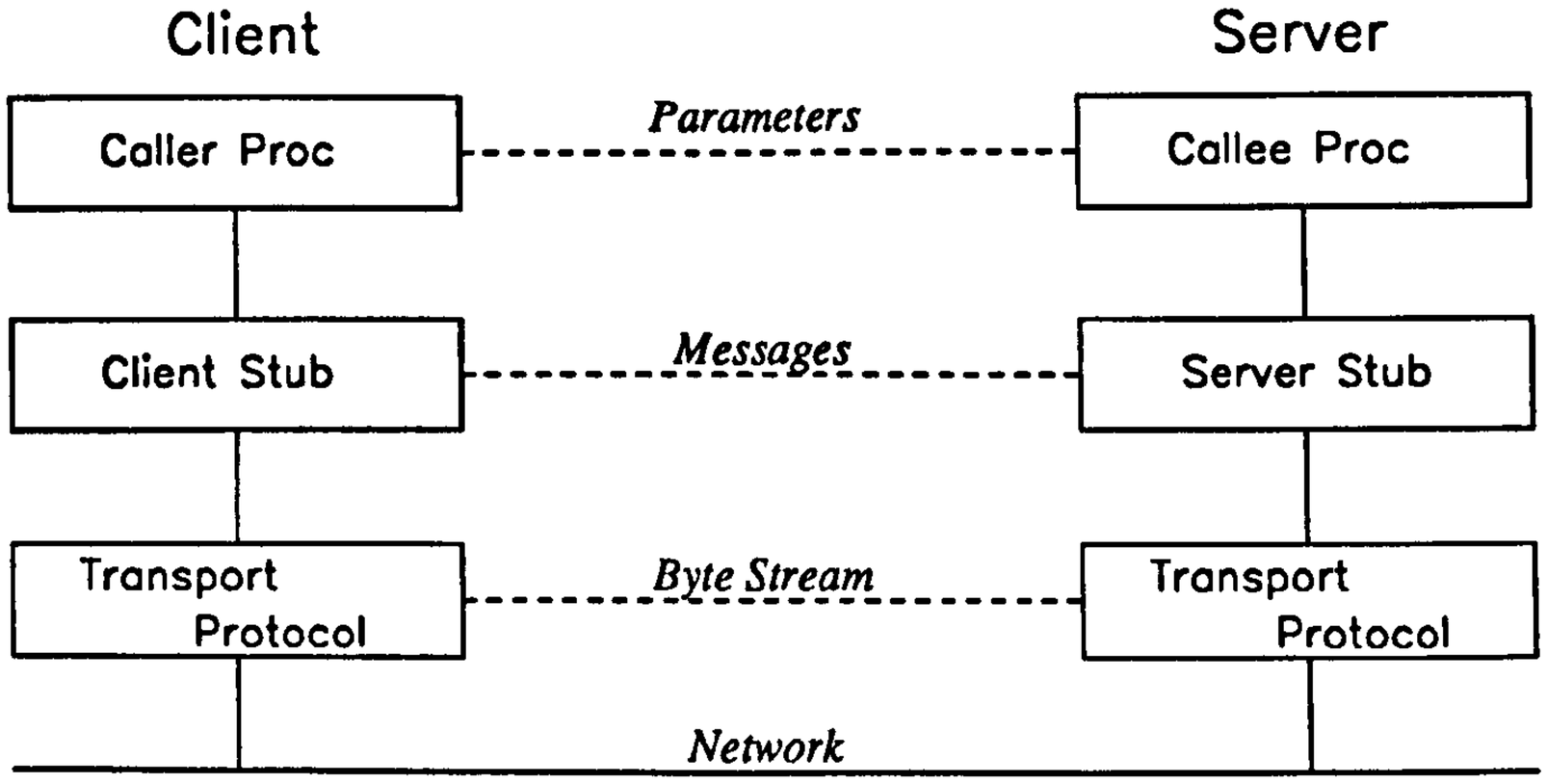

Figure 3.1: Remote procedure call mechanism

When a client program makes a remote call to a procedure in a server program, it actually executes a perfectly normal local call to a stub procedure mirroring the remote procedure at the server side. This *client stub procedure* is usually generated by the RPC mechanism from information about the remote procedure name and types of its arguments and result. The client stub packs the arguments into a message (byte array with some known interpretation) and calls the transport protocol.

The transport protocol then *reliably* sends the message to the server, using an unreliable but very efficient lower-level protocol based on packets — an uninterpreted byte array. (However, there are transport protocols that are deliberately not reliable to increase throughput; see section 3.3.7 for a discussion.) Then the client stub "blocks" waiting for the result message.

On the server side, the transport protocol level is listening to the network, waiting for incoming packets. When one arrives, it builds a message and calls the appropriate server stub passing the message as an argument. The server stub un-packs the arguments from the received message, calls the intended procedure, packs the result into a new message and passes it to the transport protocol, that sends it back to the client.

On the client side, the transport protocol is waiting for the result packet. When it arrives, a message is built and passed to the client stub, which un-packs the result value and finally returns it to the client program.

Although the client and server have been separated for presentation purposes, nothing in this design prevents clients from being servers at the same time. Indeed, this is the case when distributed object systems like Emerald [BHJ+87] need to support remote method calls in a transparent way from potentially any program in the system to any other program.

3.3.2 Application Programming Interface

There are at least two instances when the programmer interacts with the RPC mechanism: when stubs are generated (client and server side) and when they are used (only at the client, as a "dispatcher" linked to the transport protocol on the server calls the stub based on the message content).

A *stub generator* is a program that generates the stubs for the client and the server, and for this it must have access to the names of the remote procedures and the types of their respective parameters and results. This information is also known as the *signature* of the procedure and is usually defined in an *interface description language* (IDL) [BN84, WB87, OSF91, Sun93b, OMG95] (see also section 2.3.1). The stubs are generated in the same languages as those used to implement the client and the server programs so that they can then be compiled and linked in the usual way. The client stubs can then be called as any other local procedure.

One of the design goals of earlier RPCs was that the syntax and semantics of remote procedure calls should be as close as possible to those of local calls [BN84]. However, local and remote calls are fundamentally different, for example network delays or partial failures may occur (see section 3.3.9) and parameter semantics may be restricted (see section 3.3.6).

This is the reason why Hamilton [Ham84] decided to add extra information to remote calls for RPC control. Even if the semantics were the same, the fact that remote calls take 3 to 5 orders of magnitude longer than local calls — due to extra CPU costs and network latency — is a valid reason for alerting application programmers to the difference.

Recently it was recognized that if the signatures of the remote procedures are available in the program itself when the client stub is called, then there is no need to define these signatures in a separate IDL. Aiming for simplicity, Network Objects [BNOW93] is an RPC mechanism — very well integrated with Modula-3 [Har92] — that generates the stubs as part of the compilation process. The programmer still has to specify which procedures are (potentially) remote, but the separate language and its compilation and linking are no longer required.

Finally, it should be noted that the decision to generate the stubs is purely an engineering optimisation. The functionality provided by all stubs could be provided instead by a generic stub that would pack and un-pack a value of any type [HL82, BJW87, Cra93]. In a language where the type can be extracted from values at run-time and which supports dynamic binding, the stubs themselves can also be generated, compiled and linked to the application "on demand" when they are used for the first time.

3.3.3 Server Binding

Before a client is able to call a remote procedure it must know the network address (location) of the server, a process commonly known as *server binding*. Some very important

servers may have their location publicised as "well-known addresses", but the vast majority of the addresses will have to be found at run-time. This is also necessary so that services can be relocated or closed. For the client to be able to discover the addresses of the servers implementing a certain procedure, this information must be stored somewhere.

Typically the process works as follows. When a server starts running it *exports* its remote procedures to a *binding service* [Ham84, WB87] — also called a *naming service*. The client must then contact this entity to find the addresses of servers implementing some procedure, and then *import* it.

It could happen that more than one server is able to execute a procedure. The binding service can then choose the "best" server for the client or allow the client to choose one itself (e.g., the closest server, the server with the lowest process load, the most reliable server, or the cheapest).

The binding service is a potential bottleneck in the system if it does not scale well or if it, or access to it, is unreliable. Replication can be used for increasing both the reliability, availability and scalability of the binding service. But replication also introduces a replica coherency protocol (see section 2.3.6) that makes the RPC mechanism more complex to implement and introduces dependencies between stores.

3.3.4 Type-checking

For the server to make sure that the incoming procedure call is valid, some form of type-checking against the parameters must be executed by the server. Usually, compilers and linkers do this type-checking for normal procedure calls, but in a distributed environment — where client and server are built and run independently — type-checking can only be enforced at run-time by the RPC mechanism itself [Ham84].

A possible solution to run-time type-checking is for the external data representation to be *self-describing* (see section 3.3.8). In this case, the byte array sent to the server representing the arguments to the remote procedure also contains the necessary information to rebuild its own values in a type-safe manner. However, for complex types this kind of representation is expensive in both the amount of data transmitted and in the time required to pack and un-pack the parameters and results.

A more efficient alternative is to send a *remote procedure identifier* describing the interface, e.g., a fingerprint of the procedure interface as in Network Objects [BNOW93]. If the fingerprint is generated by both stub generators at the client and server side, then the server may check for each incoming message if it corresponds to a valid remote procedure. Although fingerprints are not guaranteed to be unique, the probability of error can always be made smaller by simply using a larger number of bits.

3.3.5 Call Semantics

Remote procedure calls have failures due to network problems that one does not expect to happen in local calls, thus the *call semantics* for local and remote procedure calls are necessarily different. The designer of an RPC mechanism must decide which *error recovery procedure* must be executed when a fault is detected in the network.

There are three basic approaches for remote procedure call semantics [TA90].

1. *At-least-once*—The call is attempted, and if the response is not received in a certain period of time, the message is transmitted again. This period can be configured for a server, for a service on that server, for a remote procedure or for a particular procedure call.

2. *At-most-once*—The call is attempted, and if the response is not received in a certain period of time, the client is notified of the error.

3. *Exactly-once*—The call is attempted, and if the response is not received in a certain period of time, the message is transmitted again. In this case, however, the server must keep a record of the received messages and execute the call as a transaction to achieve the "all-or-nothing" effect. (The "exactly once" semantics in [Ham84] is actually at-most-once semantics.)

The at-least-once approach may execute the remote procedure several times. If some degree of semantic transparency is to be achieved, all remote procedures must be idempotent, a result usually hard to accomplish (especially when remote procedures update a database). This is the reason why most RPC systems offer one of the other two sorts of semantics.

Exactly-once is the one which offers the highest level of protection against failures, but this would not be the best approach if the client is interested in the errors. It also requires an expensive protocol to guarantee the call is made only once, a price not all applications may be willing to pay. Finally, the client may wait indefinitely if the server fails and does not recover.

Although at-most-once does not give strong call guarantees, it offers a potentially higher performance. Knowledge of the errors may be useful for the client to try other alternatives; for example, experimenting with another server or abandoning the call. Also, the client may implement other forms of fault recovery, such as local transactions. Finally, client and server may be connected within a regime offering a higher-level of fault recovery, e.g., distributed transactions (see section 3.3.13).

3.3.6 Parameter Semantics

The semantics for parameter passing is one of the major issues of this thesis because of the richness of data types supported by modern persistent languages like Napier88 (see section

3.2.2). In this section we just give a brief overview to the topic, which is described further in chapter 5.

Parameter semantics can be broadly divided into two main topics: *how to pass* the parameters (by reference or by copy) and *what to pass* (simple types, constructed types, procedures, abstract data types, and so on).

Call-by-reference or Call by copy

Passing parameters by reference to remote procedures requires local (normal) references to be already remote (global) references or be transformed into remote references before the call is made. This has at least two advantages: large structures are not moved unless they are needed; and mutable values are not replicated so they can be shared by local and remote values alike.

Passing parameters by reference is possible only when both client and server are written in the same language because it is implemented at a very low-level by changing the compiler (or the abstract machine). For example, Wai [Wai88] implemented a distributed version of PS-algol [ABC+83] that supports call by copy for scalars but call-by-reference for pointers.

However, ultimately data and code have to be in the same address space for computation to proceed. When the argument is accessed, either the value is copied to the server or a call back from the server to the client is used. In the latter case where the value is not copied, potentially many messages are exchanged with accumulation of latency. It also makes the remote computation more dependent on the availability of the client program, as it may be needed at any time during the entire remote computation. Finally, it requires the need to manage references between machines, e.g., for garbage collection.

These are the reasons why most RPC mechanisms do not support complex data types and pass parameters by value, that is, all parameters are deep copied from the client to the server [BN84, Ham84, Lis88, Sun93b, OSF91, JSS94, Sun96a]. But migration by copy also has its problems. The transitive closure may be large, even though probably not all of the objects copied will be used in the server. Migration by copy duplicates many objects that already exist in the server, e.g., when they were copied in a previous call. This can result in the destruction of sharing semantics in the presence of multiple copies of the shared values. (Section 3.4.2 elaborates on these problems.)

Many intermediate models between these extremes also exist. For example, only the top value can be copied and all referenced objects are transformed into remote references to the local objects. Or the argument can be copied by following the transitive closure to a certain depth [KOMM93, KKM94] or until a maximum buffer size is reached [THM+96]. The distinction between what is copied and what is passed by reference can also be stated at compile-time [BNOW93, Lop96]. However, all these intermediary schemes suffer from the problems introduced both by call-by-reference and call by copy. We suspect that application programmers may have difficulty comprehending the resultant complex semantics.

(We discuss call-by-reference in section 5.1 and call by copy in section 5.2, and in sections 5.3 and 5.4 we motivate and present a proposal for a new compromise between these two models of parameter passing.)

Restrictions on Argument Types

To be as useful as possible, the RPC mechanism should be capable of transmitting the full range of data types and values in the programming language. This is restricted in practice by several constraints.

1. References to the outside world that are meaningless in other programs and computers cannot be transmitted, e.g., socket descriptors.

2. Types that do not exist in the programming language in which the remote program is written, e.g., graphic types in Napier88 (see section 3.2.2) when the call is to another language.

3. Types that exist in the remote language but for which the semantics differ, e.g., type-safe `variant` in Napier88 and type-unsafe `union` in C.

4. Types that exist in the remote language but for which there is no common representation, e.g., Napier88 procedures cannot be passed to programs written in other languages.

5. Values that create difficult implementation problems, e.g., deep transitive closures which require copying large parts of the store in Napier88 (see section 3.4.2).

6. Values of generic types — such as pointers in C or the type any in Napier88 — cannot be passed as parameters in remote procedures because the type of the actual parameter may be one of the non-supported types.

As a consequence, most RPC mechanisms only support a sub-set of the type system of the target programming languages. For example, Sun/RPC [Sun93b] is an RPC for C that only supports the passing of scalars and simple constructor types (records and unions) as arguments. Even a modern RPC like CORBA [OMG95] restricts the types of arguments in object methods to basically the same sub-set as Sun/RPC (although it adds support to pass arrays and global object identifiers).

The RPC may attempt to support rich data types. For example, one of Hamilton's goals [Ham84] was to permit the widest possible range of types while still maintaining type-safety. However, the following restrictions still apply: no support for procedure variables, no support for references to objects outside the language scope, and no support to the any data type (the union of all types). A method for transferring abstract types is proposed but not implemented. A violation of these restrictions is detected only at run-time.

More recently, a number of authors have proposed RPC mechanisms with richer parameter semantics, including the possibility of passing procedures and other code representations as arguments in remote calls.

- A typical approach is to pass procedures by reference as proposed by Kato *et al* with *Distributed ML* [OK93] and *Distributed C* [KOMM93, KKM94] or Cardelli with his new script language called Obliq [Car95a]. When one of these procedures is called, the RPC mechanism makes a *remote callback* and executes the procedure in the original address space. This approach is not appropriate for large distributed systems because it increases dependencies between machines, generates network traffic, and has no support for partial failures (see section 5.1.4).

- Another approach is to copy the procedure value itself. Examples of RPC and other distributed mechanisms based on this approach include *remote evaluation* [SG90], *remote execution* [DRV91], Facile [Kna95], Tycoon/RPC [MMS96, MMS95, Mat96] and Java *applets* [AG96]. This solution raises another set of problems—large transitive closures, loss of coherence between original and copies, and so on—described in section 5.2.4.

Migration by substitution, proposed in section 5.4, is a compromise between these two extremes of parameter passing semantics that can be used to help migrate procedures and other complex types between autonomous stores. An example application and performance measurements are presented in chapter 7.

3.3.7 Transport Protocol

The client and server stubs use a *transport protocol* for exchanging messages—byte arrays containing packed arguments and results—between programs. The transport protocol typically offers one of the following two modes of operation.

- *Connection-oriented mode*—Data is transmitted along a virtual circuit in a reliable, sequenced manner. Because setting-up a connection is an expensive operation but data transmission is then cheaper, connection-oriented communication is appropriate for bulk data transfer. However, this mode commits resources which may then be under utilised.

- *Connection-less mode*—Offers message-oriented, non-reliable transfer of data with lower latency when compared with the connection-oriented mode. However, the overhead per message is greater than the connection-oriented mode because the target address has to be transmitted in each message.

TCP is a connection-oriented protocol that automatically gives at-most-once semantics to the higher levels of the RPC, while the connection-less UDP protocol is not reliable and

gives only at-least-once (see section 3.3.5). Thus, in choosing what protocol to use for the RPC implementation, either reliability is not an issue for the application or it has to be built at a higher-level within the RPC implementation. Alternatively, the RPC may just offer an unreliable variant of remote call which forces application programmers who need reliability to implement it at the application level.

Sun/RPC [Sun93b] is implemented on top of a transport protocol called Transport Level Interface (TLI) [Sun93c] that offers both modes of service: a reliable interface based on TCP and an unreliable transport protocol on top of UDP. Many other RPC systems offer both connection-oriented and connection-less modes [TA90].

In addition to the mode of operation, the transport protocol may also support other functionalities such as dividing a message into smaller messages or buffering several messages to be sent as a single message. The decisions can be made at compile-time or dynamically, for efficiency or other reasons.

3.3.8 Data Representation

The values in a program are data structures, but to send these values across the network they need to be converted to a byte stream. For example, integers can be represented by 4 bytes, and strings by their length followed by the characters. Some sort of *external data representation* — to which both the client and server agree — must be defined, and algorithms to convert into and from that representation designed and implemented (eventually for a number of languages).

But just a raw *data representation* is not enough. There are many reasons why the *data format* from a source may differ from the data format at a target, e.g., to accommodate inter-working with other languages and different machine architectures (see section 2.3.7). There are two main approaches to communication between a client and a server in heterogeneous environments described below: common data format and source data format.

Common Data Format

With the *common data format* all transport protocols use the same data format for communication independently of the native format of each program. This approach is simple and suitable for general heterogeneous environments, where there can be a large and extensible number of programs, each with its own data format.

XDR [Sun93a] is an example of the common data format proposed by Sun that has become a *de facto* standard. It has been used extensively as part of several operating systems, as well as in a number of RPC mechanisms [BCL+87, Gib87, Sun93b]. XDR works as a *lingua franca* and makes them all compatible with each other, provided they restrict themselves to the types supported by XDR and use the same control messages.

ASN.1 is an ISO standard and another example of a common data format that, unlike XDR, supports a notation for defining the type alongside each item in the byte stream. It should be noted that if client and server agree beforehand on the type of the message, then this type information is still sent even if redundant.

Source Data Format

With the source data format all transport protocols pack values in their own format together with a format identifier, and every transport protocol knows about all other data formats in order to un-pack any incoming message. This approach has the advantage of potentially better performance because in many situations the source and target use the same data format, so no redundant information about types is needed. It is, however, much less flexible and does not scale well to support a large number of formats.

In contrast to the common data format of Sun/RPC and others, DCE/RPC [OSF91] has opted instead for the source data format. DCE/RPC allows distributed applications to run over heterogeneous environments by transferring messages tagged with a description of the basic data representations of the calling machine. DCE/RPC claims that great efficiency gains can be achieved in this way if both the client and the server are written in the same language and execute over the same machine architecture.

If it is intended for the RPC to run on many architectures and support a large number of target languages, then the source data format of DCE/RPC has a two-dimensional problem: for L languages and M machine architectures, $(L \times M)^2$ conversion procedures must be written. Even using a common data representation this number of different implementations is still L x M. A configurable stub generator based on languages and machine specifications has been proposed to solve this problem [Gib87].

3.3.9 Failure Model

While it can be argued that the syntax and semantics of a remote procedure call do not need to be different from a local one under normal executing conditions, problems arise when the client, the server or the network are subject to failure. A *failure model* specifies how to detect these failures, group them in some meaningful categories and report them to the calling program in an understandable and useful manner. Exceptions at the language level are particularly well suited for dealing with failures during remote calls, because they separate failure treatment from normal application execution.

It may be useful to distinguish between errors in the network, in the remote machine or in the remote application. For example, a deadlock in the remote server can trigger a timeout in the client which can be easily confused with network congestion. However, it is a well-known problem in distributed systems that this information cannot be easily obtained.

This is one of the reasons for the popularity of RPC systems with support for transactions, such as the one provided by Encina RQS [ESS91, Tra93b]. A *transactional RPC* coordinates the remote call, data migration and remote operation as a single atomic action that either succeeds or fails completely (see sections 2.3.8 and 3.3.13). Thus if a failure occurs when a remote call is being executed, the distributed system will not be left in an undefined or inconsistent state.

3.3.10 Asynchronous RPC

An asynchronous RPC mechanism—sometimes called message passing—calls a remote procedure without blocking to wait for the result. This permits a client to call several remote procedures concurrently and has been argued as a basis for exploiting the natural parallelism found in distributed systems [ATK92a]. Asynchronous RPC can also be used so that the client performs other (local) operations while the server is computing the remote procedure.

Asynchronous RPC can be implemented with a normal blocking RPC mechanism and threads. For each asynchronous call, a thread is created and a synchronous RPC performed. However, this solution does not scale well for a large number of parallel remote calls and is otherwise a complex solution for an operation with such simple semantics [ATK92a].

Instead, native asynchronous RPC has been proposed together with language support. Athena/RPC [SM86] provides a non-blocking asynchronous mode that was developed for improving performance when *no result* is returned from the procedure. *Stream* in the MIT Mercury system [LBG+88] combines synchronous and asynchronous bulk data transfer in a clean and uniform way. Streams have been also integrated into Argus [Lis88] as a new data type called *promises* [LS88]. Futures [WFN90] are very similar to promises, but were designed instead for low latency (not bulk data transfer).

3.3.11 Heterogeneity

RPC is more flexible if the client and server programs can be written in any of a number of programming languages and execute in machine architectures with different data formats. This has been discussed already in section 3.3.7 in the context of the transport protocol. However, heterogeneity has implications that exceed those handled by the transport protocol.

- *Application programmer interface*—The API for users of the RPC system has to be independent of a particular programming language.
- *Data types supported as arguments to remote procedures*—Because each language supports a particular set of base and constructor types, a compromise is required between support for heterogeneity and type-completeness (see section 3.3.6).

A number of RPC mechanisms have been designed and built with heterogeneity in mind, including Athena/RPC [SM86], multi-language RPC [Gib87], heterogeneous RPC [BCL+87, BLL+88], DCE/RPC [OSF91] and Sun/RPC [Sun93b]. Some of these use a common data format (see section 3.3.8) that can be used to achieve inter-operability between RPC systems themselves. Heterogeneity is also one of the main motivations for CORBA [OMG95].

3.3.12 Performance

RPC performance is critical because remote calls are typically 4 to 5 orders of magnitude slower than local calls [WWWK94]. On the other hand, experiments have suggested that the large majority of remote calls for some kinds of application are intra-machine calls, i.e., are made between programs executing on the same machine [BALL89]. As a result, a number of optimisations can be made for this common case, e.g., no data translation to a common data format is necessary and communication using shared memory can be used instead of sockets.

Lightweight RPC [BALL89] is an RPC mechanism that optimises intra-machine calls on a shared memory multi-processor. A number of optimisations are employed to achieve a higher call throughput and lower latency yielding a factor of 3 in maximum performance improvement when compared with other well-known RPC systems. Schroeder and Burrows [SB89] and more recently Liedtke [Lie93] have shown that impressive performance improvements can be achieved if the operating system itself is designed for high-performance IPC.

3.3.13 Transactional RPC

An RPC system can be used to update a remote database or perform any other destructive operation for which its success (or failure) is crucial for the application. Unless all remote operations are idempotent, this is always the case.

If the communication is reliable, then the simple act of returning a result to the client will confirm the operation. However, operations can be performed and the result lost, e.g., the server may crash immediately after processing the remote call and before sending the result back. Thus the client can never be sure whether the call has succeeded or failed, completely or partially.

A *transactional RPC* attempts to solve this problem by executing the remote call as a transaction that can be coordinated with another transaction executing locally at the server. For example, Argus [LS83, Lis84, Lis88] integrates both transactional servers and reliable RPC so that remote operations either entirely succeed or it is as if the remote call never happened. This behaviour can be generalised to several calls in a sequence.

More recently, transactional RPCs have been offered as (or integrated into) commercial products. For example, Encina [Tra93a] provides a transactional RPC [ESS91] to support reliable remote updates. It should be noted that a transactional RPC is a particular case

of the distributed transaction model with only two participants. (A *transaction monitor* like Encina is to be used in the general case, e.g., if process `A` calls procedure `p` in process `B`, then procedure `p` calls procedure `q` in process `C`.)

3.3.14 Object Orientation

In an object-oriented language there are objects with hidden data and visible methods, instead of inter-connected (but mostly independent) data and procedures. In a *distributed object-oriented language* all programs in the distributed application can use objects in any other program by (remotely) calling its methods.

A method operating on a remote object works like a remote procedure where the first parameter is the object identifier. The system is responsible for keeping the location of the object and for generating the necessary stubs.

Emerald [BHJ+87] is a distributed object system that also supports object migration between servers. In Emerald, programmers write distributed applications without worrying about object location; objects move between servers for availability or performance. (However, programmers can control object migration if they want, see section 5.1.2.) Network Objects [BNOW93] also supports methods that can be called transparently between programs. CORBA [OMG95] is also based on an object model for remote invocation.

3.3.15 Extensibility

There appears to be a wide range of possible semantics associated with remote procedure call: support for object migration, replication, distributed transactions, persistence or performance. Not only are many of these semantics incompatible between each other (e.g., high-performance with distributed transactions) but there are also many ways to implement them. This may suggest a remote procedure call mechanism should be designed to support an open set of extensions to incrementally accommodate more features as necessary.

Subcontract [HPM93] is an RPC system that eases the task of integrating several RPC features and the incremental addition of new mechanisms in a compatible way. Extensions are possible due to an "operations vector" that is used by the stubs when executing a remote object call. In this way, the application programmer gains control over the basic mechanisms for calling a remote procedure without changing the basic RPC architecture.

3.3.16 Conclusion

This section has presented the RPC design issues most relevant to this thesis. There was no intention of describing RPC completely and even less to give a tutorial on RPC. Namely, the following highly advanced or not directly relevant RPC issues have not been addressed

in this section: multi-cast and broadcast, orphan treatment, security and authentication, naming and binding, and server management in general. (Though naming and binding are discussed in chapter 4 in the context of our type-safe RPC.)

More information on RPC can be found in the publications cited above or in the following general references: a taxonomy of RPC [Spe82]; RPC design issues [WB87]; a survey of RPC [TA90]; and general books on distributed systems [Mul93, CDK94].

3.4 Combining Persistence and RPC

In the previous two sections we have presented persistence and RPC. We now analyse how the high-level approach to programming provided by persistence introduces new possibilities and expectations — but also important difficulties — to RPC design and implementation. (Chapters 4 to 6 will revisit these issues in detail and propose some solutions to the challenges presented below.)

3.4.1 Opportunities

The benefits of persistence for application development in general also apply to RPC construction. For example, chapter 4 explains how a type-safe persistent RPC was developed in 3 months by the author alone. This RPC has since then been continuously evolving, which also gives an idea of the support for incremental construction of persistent applications.

Here we repeat the most important features of Napier88 described in section 3.2.1 to explore how they may help with RPC design and implementation.

Orthogonal Persistence

Orthogonal persistence simplifies programming whenever long-lived data is required by the application. An RPC implementation requires data: general auxiliary data such as import and export tables, client and server stubs (with first-class procedures, see below), and run-time information such as partial results and cached data that survive program execution.

It may also help if the programmer developing the RPC has access to the language implementation. Persistence needs procedures that write data to, and load data from, the store. These procedures can be modified to write to, and read from, a socket connection (see, for example, [Mun93]). However, these procedures are language and implementation dependent and cannot be used for communication between persistent programs written in different languages.

Rich Type System

The rich type system offered by advanced persistent languages like Napier88 creates the opportunity for the RPC to transfer more interesting data structures because communication now is not restricted to scalar types and simple constructed types. For example, procedures and abstract data types can now be exchanged between client and server. (The rich type system is also where the major challenges reside; see section 3.4.2.)

Great benefits from working with a rich and reflexive type system can also be achieved at the application programmer interface, as type enquiries obviate the need for a separate "interface definition language".

Strong type-checking in a persistent language can be based on structural equivalence. In Napier88, for example, each type is represented by a value that can be transferred and compared with other type values in a remote store (assuming they are running the same version of Napier88). This facilitates type-checking in a distributed persistent environment.

Dynamic Binding

Napier88 permits new objects, including procedures, to be created and dynamically bound to the current execution in a type-safe manner. This permits client and server stubs to be created and changed at run-time without the need to interrupt the program for linking with a separate library. Thus persistent programming languages and persistent RPC are suitable for distributed environments where continuous operation is required.

First-class Procedures

Both client and server stubs are implemented as procedures. Because in Napier88 procedures are first-class citizens, stubs do not need to be written to a file, then compiled and linked separately. In conjunction with reflection (see below) stubs can instead be dynamically generated and added to the store. Persistence allows the stub generation cost to be conveniently amortised over many program executions.

Language Reflection

Reflection is the capability to augment the program with new code at run-time [Kir93]. Reflection in an RPC can be used, for example, to support the creation of new stubs at run-time (in conjunction with dynamic binding and first-class procedures).

Concurrency

Servers can take advantage of Napier88 threads to accept a number of remote calls concurrently and service them in parallel. Concurrency can also be used to simulate asynchronous RPC by creating a thread for each asynchronous call that makes a normal blocking RPC while the client continues executing.

3.4.2 Challenges

We now briefly present the main challenges introduced by combining persistence and RPC.

Rich Type System

The RPC should permit objects of any type as parameters and results in remote calls. Passing scalar types and simple constructor types, including any shared and cyclic data structures, is now well-understood [HL82, BJW87, Cra93]. But in addition to these types, Napier88 also supports first-class procedures, abstract data types and infinite unions. Although techniques for passing values of these advanced types now exist [Kna95, MMS95, BC95b], they are still very restricted in their scope and not well integrated with other mechanisms for distributed computation.

Type-safety

Strongly type-checked languages are helpful in achieving correctness. System facilities such as RPC have to sustain the type-safety for application programmers that use these systems. In a distributed application, however, many parts of the application are built and changed independently, creating more opportunities for inconsistent types between programs.

Another problem arises with the RPC implementation itself. Although most of the RPC code can be written in the type-safe language, access to non-safe language constructs is necessary to write an RPC system (see below). For example, in a version of Napier88 — available only to its implementors and for this research work — there are special low-level procedures to manipulate object pointers directly in an unsafe manner. These unsafe procedures are used by the RPC, for example, to build cyclic data structures. The RPC implementor has to guarantee that, despite the store passing through temporarily invalid conditions, it is correct when control is returned to the application program.

In addition to these problems, low-level (unsafe) procedures are likely to vary substantially between persistent languages, between different implementations of the same language, and even between versions of the same implementation (e.g., the traditional abstract machine (PAM) [CBC+90a] and the new PamCase [CCM95]). This complicates the task of implementing the RPC mechanism compared with an unsafe programming language such

as C which provides these operations as standard at the language level. An interesting research issue is to identify an API to give access to these features while minimising the possibility for errors and variations across different implementations.

Large Transitive Closures

Objects in a persistent system tend to be highly inter-connected, a consequence of orthogonal persistence and a rich type system. For example, the transitive closure for *persistent procedures* in Napier88 is typically the entire store. (This problem has been solved in the new PamCase implementation [CCM95].) Many objects with complex data types have the same problem; for example, those that use procedures such as the *map* implementation in the Glasgow Libraries [CAL+94]. (This is in contrast with traditional, non-persistent languages in which transitive closures are always limited by the address space of the currently executing process.)

Large transitive closures create problems for passing parameters. Typically, the parameter passing semantics for conventional (i.e., non-persistent) RPC is based on call by copy, but this is difficult or even infeasible in a persistent RPC because large parts of the store would have to be copied. (See the next two challenges for more problems exacerbated by large transitive closures.)

On the other hand, passing parameters by reference or partial copy — where part of the reachable data is copied and the rest is passed by reference — creates dependencies between stores. We will return to this discussion in chapter 5.

Preserving Sharing Semantics

We have described above why passing parameters by copy in a persistent RPC may copy large parts of the store to the target. But even when only a small part of the store is copied, passing parameters by copy creates *duplicates* that will then eventually diverge creating inconsistent copies of the same object. Worse, *identity checks will fail* where a programmer might reasonably expect them to succeed because each copy has its own (local) identity. The duplicates not only destroy object sharing but with a persistent RPC the duplicates accumulate across program executions. On the other hand, a strict replication protocol only re-introduces the kind of problems created by remote references. All these problems are amplified by large transitive closures.

Sharing Objects Between Stores

In order to retain the simplicity of a single store in a distributed environment, the RPC would ideally support object sharing between stores based on one of two main approaches [DC93, DPS+94, Dan95]:

1. *One shared copy*—If one shared copy is chosen, then all stores (except the store maintaining the object) have to use a remote reference to access the shared object. The problems are similar to passing an argument by reference in a remote call (see section 5.1.4). Namely, *the time to access the object is several orders of magnitude more than that of a local access* as a result of network latency and other communication costs. More importantly, dependencies are created between stores. Because the remote store where the shared object resides or the network connection may fail, *referential integrity can no longer be guaranteed for persistent shared objects.* This violates one of the fundamental features of orthogonal persistence.

2. *Many replicated copies*—The only option that minimally disrupts persistence is to replicate the object to the stores where it is used. However, replication requires a *coherency protocol* (to propagate updates before values in replicas are used) that in turn introduces remote references. Depending on the access pattern to the object (the number of reads compared with the number of writes) the benefits of replication may be easily outweighed by the network traffic and latency introduced by the coherency protocol.

Partial Failures

In a distributed environment the computation no longer depends on a single system: remote programs may crash, computers may stop and the network may be slow or disconnected. This is in contrast with the failure semantics offered by a local persistent store, where it is either working normally or has completely stopped.

Partial failures due to distribution have to be introduced into the local computation model in a way that can be understood and dealt with by application programmers. Finding a good model for detecting, reporting and explaining these—previously non-existent—partial failures to the persistent application programmer is a difficult research issue.

3.4.3 Need for Compromises

In a distributed context it is very difficult, or even impossible, to maintain the abstraction of a uniform store sought for orthogonal persistence while still being realistic. For example, we cannot ignore partial failures and still attempt to support object sharing with referential integrity between stores. Objects cannot be shared amongst many stores efficiently and reliably, but we also cannot replicate them if they have large transitive closures. Attempts to solve one problem only make another more visible.

Compromises are needed, but any relaxation in the uniformity of orthogonal persistence must be carefully considered. The new semantics should be close to the local persistent semantics for two reasons: to be easily recognized, understood and accepted by existing persistent programmers; and to accommodate any techniques and tools designed for local

programming that should still work locally and be adaptable for distributed environments. Finding these compromises is a goal of this research.

3.5 Summary

This chapter introduced orthogonal persistence and Napier88, a research persistent language used for the experiments described in this dissertation. We then presented those RPC design issues that are relevant to RPC in general and in particular to RPC in a persistent environment. We were then in a position to analyse what happens when persistence and RPC are combined.

We conclude that RPC is a simple yet powerful communication model that may be used by persistent application programmers not familiar with distribution in order to build distributed persistent applications. We suspect persistence and RPC are an interesting combination, although it also creates new and difficult problems. Both the possibilities and the problems will be explored in this thesis.

Chapter 4

Type-safe Persistent RPC

This chapter describes an RPC mechanism built in Napier88 that automatically guarantees strong type-safety between a client and a server. The chapter starts by explaining why type-safety is important and introducing techniques to enforce it. The design and implementation of the RPC is then presented, together with a description of the interface to the programmer and a complete example.

4.1 Type-safety

When calling a remote procedure one cannot assume that the types of the arguments at the client will always match those of the actual remote procedure at the server. This is an inevitable consequence of autonomy in distributed systems which allows components to change independently.

Type-checking must be enforced in order to prevent programmers from making errors, particularly those which might endanger the integrity of the whole system. Integrity is especially relevant in the context of persistent systems; a server crash may corrupt the database and the error may become manifest much later (possibly months after the call has terminated).

4.1.1 Example of The Problem

The traditional RPC is not type-safe, as a simple experiment using Sun/RPC [Sun93b] demonstrates.

A procedure for remotely printing an error message is initially declared as `pe(int)` and later changed to `pe(string)`, e.g., to change from an error code to an error string. If the programmer at the server updates the procedure argument type but the client is not

changed accordingly, next time the client calls the procedure the server simply crashes. The crash is a consequence of the server trying to use as a string a value that is in fact an integer.

This type mismatch and consequent server crash are easy to produce because Sun/RPC only *assumes* that both the client and the server use stubs generated from the same procedure signature, without actually *enforcing* that assumption.

In order to deal with evolving remote procedures, Sun/RPC offers "protocol versions" so that servers have the opportunity to change the specification by adding a new signature version. The client would continue to use the old version as long as required, then use the updated specification to generate stubs for the new version. This framework may help, but as we have demonstrated with the experiment above *safety is not guaranteed.*

4.1.2 Type-checking

The safety problem has been recognized before. For example, Network Objects [BNOW93] is a modern RPC well integrated into Modula-3 [Har92] that permits object methods to be called remotely across the network. Great care was taken with respect to type-safety; because the native "type codes" generated by the Modula-3 compiler are valid only within the local program, the compiler had to be changed in order to compute a fingerprint for every object type. Two types have the same fingerprint only if they are structurally equivalent.

In a type system based on *structural type equivalence*, two values have equivalent types if the types have isomorphic structures [ADG+89]. Structural type equivalence contrasts with *name type equivalence*, in which two values have the same type if the types share the same type declaration. Only structural equivalence offers a concept for type equivalence independent of a particular type declaration in one program, that is, for separately compiled programs [BHJ+87, ABM88, CBC+90b, Con91].

Thus Network Objects prevents method calls on objects with incompatible types even between independent programs. (Strictly speaking, there is a low probability that two different types will have the same fingerprint. However, this probability can be made sufficiently small that it can be treated as negligible.) Previous RPC mechanisms that enforced type-safety were more restrictive because they depended on name equivalence, thus requiring that equivalent types be the result of the same compilation, e.g., Hamilton's RPC [Ham84].

Our type-safe Napier/RPC uses structural type equivalence for detecting compatibility between types declared in the client and the server. This is possible because the Napier88 compiler arranges for information about types to be recorded with the values (types **any** and **env**) or knows the type of each value at compile-time (all other types) [Cut93].

Using this type information, Napier/RPC is then able to:

1. enquire about the type of a value at run-time (using an existing Napier88 procedure called `getType` available in the Standard Library [KBC+94]);

2. exchange this type between programs (type representations are first-class values in Napier88); and

3. compare their values for equivalence (using a procedure called `EqualType`).

Fingerprints could also have been used in Napier/RPC as accelerators or reasonable approximations. However, they would have to be computed, whereas Napier88 already provides mechanisms to manipulate types and compare them for equivalence.

4.1.3 Server and Procedure Binding

Before a client can call a remote procedure it must know which server supports the desired procedure (server binding) and within that server which procedure it should call (procedure binding). Information to identify the server can be provided at compile-time by giving a network address alongside the procedure signature or delayed until later at run-time before calling the remote procedure. A similar mechanism is needed to identify the remote procedure at a server.

Many RPC mechanisms shift the responsibility for both server and procedure binding to application programmers, who must agree on names or numbers to identify the remote procedure and provide server addresses. This is sometimes called *manual binding*. For example, Sun/RPC [Sun93b] uses numbers for procedures and machine names for servers. Hamilton [Ham84] extended the syntax for calling remote procedures to accept a server address, whereas procedure binding uses "remoteproc identifiers" (64 bit unique numbers) generated by the compiler.

Having this information embedded within the signature of the procedure or in the client code may reduce the scope for future change, e.g., to move a remote procedure from one server to another. Also, low-level approaches to procedure identification (such as numbers) force client and server programmers — who should be able to build and change local programs independently — to agree and synchronise with respect to these numbers. Agreements like this are difficult in a large-scale application in which there are many programmers involved. It is also *not type-safe* because there is only an agreement between programmers which may not be respected.

On the other hand, Network Objects [BNOW93] provides *automatic binding* because the system knows where to direct the call (the remote reference to an object includes its location). Some other mechanism such as a *binding service* (see below) will have to be used to bootstrap the system, i.e., to get the first remote reference.

A *binding service* is a special server that stores the signatures of remote procedures and knows which servers support which procedures. A binding service not only supports both

server and procedure binding but may also be used to enforce type-safety if it also controls the right to call them.

For example, Birrell and Nelson [BN84] use a distributed database to store server addresses and the names of their supported remote procedures, but they do not attempt to check argument types. The ANSA architecture provided a "trading service" [DH93] responsible for passing information from servers to clients that includes a "type conformance service". CORBA [OMG95] provides a similar service.

A binding service was also chosen for our type-safe RPC as it offers a good compromise between the security of enforced type-safety and the flexibility provided by dynamic server and procedure binding. The design and implementation of the binding service are described in the next section.

4.2 Binding and Type-checking

Static or dynamic type-checking at the language level may be used to ensure that both a client and a server are type-safe locally (that is, within each of them). However, as any client and any server in a distributed environment should be allowed to be built and changed independently, one cannot easily guarantee that type mismatches will not occur between them.

In this section we describe a solution to the safety problem between a client and a server that also supports both server and procedure binding. (The interface to the mechanism will be presented later in section 4.3.)

4.2.1 Type Sessions

A *type session* describes the relationship between a client and a server as far as the signature of a remote procedure is concerned. A signature for a remote procedure is *agreed at the start of a session* and maintained until the end of that session. The remote procedure can then be called multiple times within that session, so the cost of signature matching is incurred only once per session, not per procedure call. Programmers may choose session lengths, as clients or servers can terminate a session at any time.

Sessions are intended to be long-lived. They are made persistent and thus become unrelated to the execution time of either clients or servers. Information about sessions just persists across program invocations automatically by virtue of orthogonal persistence. (Although both client and server need to be executing for a remote call to succeed.)

Each client can have multiple sessions running in parallel. There is one session for each remote procedure the client has successfully called and the session remains active until it is explicitly terminated. The maximum length of time for a session is the period between

the server starting and stopping support for the remote procedure.

There is no problem if a session ends *during* a procedure call because the signature has already been validated. The next remote call will just start a new session or fail if the procedure has been removed by the server in the meantime.

4.2.2 Capabilities

Type sessions are implemented with capabilities: before calling a remote procedure the client should own a capability for it. The capability is valid as long as the server supports that procedure and the client keeps the capability.

Capabilities are well known in the distributed system community, for example, they are a basic concept in the Amöeba operating system [MT86]. A capability is the concatenation of a *service identifier*, a *rights field*, and a *password* (to prevent users from predicting or forging capabilities). Capabilities are only created by a trusted entity, then given to other untrusted entities in the system that may pass them to other entities.

In our RPC the binding service is considered to be the only trusted entity in the system, and both clients and servers are untrusted. The binding service creates a capability, for each remote procedure, that is used both for *identifying* (procedure binding) and *validating* (type-checking) remote calls. The rights field is not used at present as the only operation that can be performed on a remote procedure is "call", and the right to request an operation can be made implicit by owning the capability.

4.2.3 Binding Service

The cost of session set-up can be reduced and its flexibility increased by maintaining the relevant information about remote procedures in a binding service. In our RPC, the *binding service* stores all server locations and all signatures of the remote procedures that the servers support.

The binding service works as depicted in figure 4.1.

1. Before a server starts supporting calls for a remote procedure it *exports* the signature of the remote procedure together with the server's identity to the binding service.

2. A client, before calling a remote procedure in a server, *imports* the right to call it from the binding service by providing a procedure signature. If the signature is supported by a server, the binding service returns the address of that server (server binding) and the capability. The capability represents both the identification of the procedure in the server (procedure binding) and an authorisation to call it (type-safety).

3. The client can then call the remote procedure using the capability as many times as needed, without ever requiring to contact the binding service again. (Thus the capability, stored by the client stub, implements the session.)

The above protocol makes it impossible for a client to call a remote procedure without proving before that it knows the procedure's name and the correct number, order and types of its arguments and its result. (As happens with all other schemes based on capabilities, a client could also ask another client for the capability and avoid talking to the binding service. This possibility, that does *not* break the safety provided by the session mechanism, will not be explored in this dissertation.)

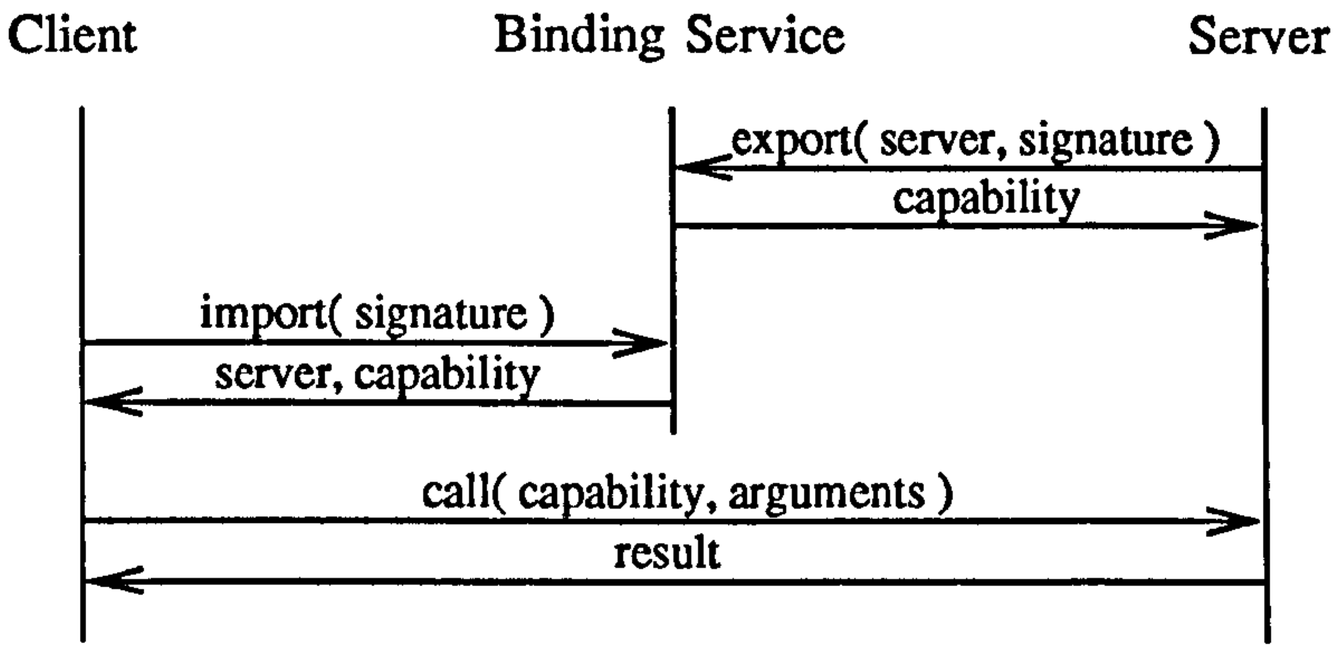

Figure 4.1: The binding service in action

The binding service is implemented as an RPC server that supports two *special* remote procedures: export and import (see example 4.1). These procedures are *special* in the sense that they have well-known capabilities, otherwise the RPC could not bootstrap.

```
type Capability is int
type Store is structure( machine: string; path: string )
type Signature is structure( str: string; rep: TypeRep )
type ServerStubName is structure( host: Store; cap: Capability )

export: proc( server: Store; sig: Signature -> Capability )

import: proc( sig: Signature -> ServerStubName )
```

Example 4.1: Export and import

As outlined in example 4.1, *export* receives as arguments a server address and a procedure signature, and creates and returns a new capability. The binding service maintains a local database to associate capabilities with their corresponding servers and procedures. If the binding service receives a signature from a server which was already previously exported by another server, it removes the previous reference and returns a new capability. (It could, of course, be programmed to remember all servers offering the presumed equivalent service and then choose between those servers when the service was requested.)

On the client side, before calling the remote procedure, each client stub checks if it already has the capability. If not, the stub itself calls *import* with its own signature, receiving the capability and the address of a server supporting the procedure, and only then calls the remote procedure. Later, the stub would discover it already has the capability, so it is unnecessary to import the procedure again.

When a client receives a capability to call a remote procedure at a server, it effectively starts a new session. This session will last as long as the server supports the procedure. When the server stops supporting the procedure, the client will note this because its next call will fail. The client can also terminate a session by throwing the capability away and freeing its resources, but this will not free any resources in the server because other clients may have started sessions on the same remote procedure.

4.2.4 Server Evolution

Each capability permits a remote procedure to be called with a specific signature at a specific server. This allows a server to change a procedure's implementation without forcing clients to start new sessions as long as the signature itself does not change. The RPC mechanism simply replaces the old server stub with a new one that calls the new implementation of the remote procedure. The client does not notice the change because the capability remains valid.

The RPC mechanism hides this kind of change from clients by default so that server programmers can decide whether an internal change in a remote procedure maintaining the signature is actually noticed by the clients.

The server can always force the clients to notice the change by stop accepting the capability for that procedure. This is made in three steps:

1. revoking support for an existing procedure;
2. requiring the binding service to delete any reference to it; and
3. starting support of a new remote procedure with exactly the same signature as before.

The client will notice that the procedure implementation has changed when a call made with the old capability fails but, when the client asks for the same procedure, the binding service returns a new capability. (Revoking support for an existing procedure is not implemented but it would be trivial with an additional remote procedure at the binding service called `remove` that accepts a `Signature` as an argument.)

Another typical change occurs when a server starts supporting a new implementation of the remote procedure with a different signature. The server should not stop supporting the old version immediately as this would force all clients using this procedure to stop calling it immediately. Instead, the server should support both the old and the new versions for

some time, allowing each client to migrate from the old version to the new one should they want to (some clients may not want to move from the old version to the new one).

Even more interesting is to integrate these two kinds of incremental change described above: new implementation maintaining the same interface and new interface with new implementation. Figure 4.2 illustrates the following example.

1. In step 1 the server creates and exports a print error procedure with a signature `pe(int)`. Only after importing the capability `cap1` will the client be able to call the remote procedure.

2. In step 2 the server creates a new version of print error with signature `pe(string)`. Initially, all clients still own `cap1` so they will continue to use `pe(int)`. But as `pe(string)` is now also supported at the server, each client decides when it is ready to start a new session in order to change over to the new signature.

3. The server administrator may then realize that many clients are still using `pe(int)` and have no intention of changing to `pe(string)`. So in step 3 the server changes the implementation of `pe(int)` to incorporate (part of) the functionality of the new procedure `pe(string)` — for example, writing the message to a log file — without the clients noticing the change. This way all clients will now use the most recent procedure implementation, even though some are still accessing it via the old signature.

4.2.5 Summary

This section has presented the design of an automatic type-safe RPC.

- The RPC is *type-safe* since calling a remote procedure is only possible after the client has demonstrated it knows the correct types and order of the arguments and result of the remote procedure.
- The type equivalence test is *strong* because the mechanism checks the actual type structure, in contrast with user-defined names or numbers that can lead to errors if used across programs built independently.
- Type-safety is *automatic* because it is achieved with type sessions managed by the RPC, without requiring any user intervention in excess of that already needed for calling a normal procedure in a type-safe language like Napier88.

The next three sections describe an implementation and how it can be used by the application programmers. In order to concentrate on the fundamentals of type-safe RPC, we omit error handling and also made no effort to optimise the use of RPC in the example applications.

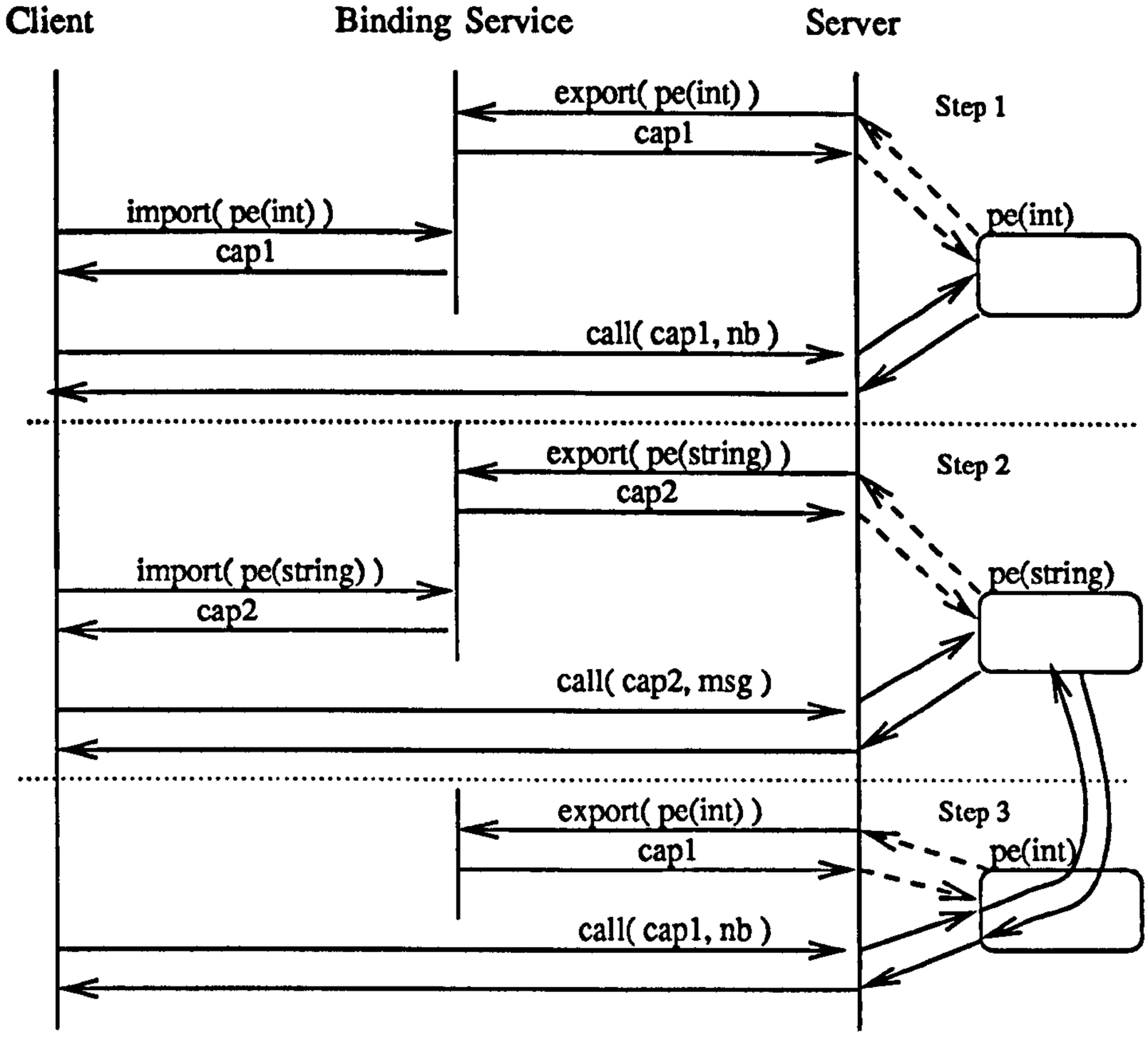

Figure 4.2: Type-safe evolution of a remote procedure

4.3 Application Programmer Interface

The interface characterises how the programmer actually interacts with the mechanism. This includes both *generating the stubs*—how to pass information about the signatures of the remote procedures to the stub generator—and *using the stubs*—how the programmer uses the client stub from within the language and implicitly uses the corresponding server stub.

4.3.1 Generating the Stubs

Programmers must provide the *stub generator* with procedure signatures—names of the remote procedures and types of their arguments and results—in order to generate the stubs. This is usually achieved by writing the signatures in an "interface description language" (IDL) used as input to an *external* stub generator (see section 3.3.2). Birrell and Nelson [BN84] described this approach and it is still used by modern RPC systems, including Sun/RPC [Sun93b], CORBA [OMG95] and Java IDL [Sun96b].

In contrast, our stub generator is *internal* and uses signatures of procedures written in Napier88 itself. This is because Napier88 has the relevant facilities (see list below) to gen-

erate and compile stubs dynamically and link these to the current execution. An *internal* stub generator has the advantage of avoiding increased complexity of the programming environment; application programmers no longer need to learn a new language and are relieved from maintaining consistency between types in procedure declarations and the corresponding IDL statements.

Four facilities of Napier88 enable *internal* stub generation (see section 3.2.1).

1. *Type system* — This permits type information to be extracted from certain constructs from which stubs can be generated at run-time (see section 4.1.2).
2. *Callable compiler* — This allows compilation of these generated stubs in a type-safe manner during execution.
3. *Higher-order procedures* — As procedures are first-class values in Napier88, a map indexing capabilities to server stubs can be constructed so that remote procedures can later be called by the transport protocol (see section 4.5.2).
4. *Orthogonal persistence* — Orthogonal persistence enables this map to be preserved between executions of the server. These two last facilities avoid the need for a separate linking process before starting the client and server.

Examples are shown below of the creation of client and server stubs and their use. This is all achieved within the language during the normal execution of Napier88 statements.

Generating Client Stubs

Example 4.2 shows how a client creates stubs for interacting with an example message server. The purpose of the message server is to store messages in a database, each indexed by an integer generated by the server. In the server, `putMsg` stores a new message returning its identifier and `getMsg` accepts an identifier and returns a previously stored message.

The client stub generator is a procedure called `makeClientStubs` that works as follows.

1. Accepts a value of type `env` (see section 3.2.1) that contains a set of procedure bindings. These bindings contain their names, their types, their values and their constancy — all the information required to generate the stubs.
2. Obtains their signatures, then generates and compiles the respective client stubs.
3. Replaces the dummy procedures with the compiled client stubs.

Writing the dummies requires roughly the same amount of work compared with writing IDL interfaces. However, in our RPC the application programmer is using the same language, and inconsistencies are avoided.

```
! create a new empty environment
let ClientProcEnv := environment()

! insert the signature for putMsg
in ClientProcEnv let putMsg :=          ! this procedure value will
    proc( s:string -> int )             ! be overwritten by the
        uninitialised[int]("putMsg")    ! client stub generator

! insert the signature for getMsg
in ClientProcEnv let getMsg :=          ! this procedure value will
    proc( i:int -> string )             ! be overwritten by the
        uninitialised[string]("getMsg") ! client stub generator

! construct the client stubs for all signatures
makeClientStubs( ClientProcEnv )
```

Example 4.2: Generating client stubs

After calling `makeClientStubs` the programmer may immediately utilise the environment `ClientProcEnv` that now contains the client stubs or place it in the store for later use.

By default, all client stubs are created with a well known invalid capability (e.g., with value 0). When a client stub is called, it first checks if its capability is invalid. If it is, it imports a capability and a server address from the binding service (see section 4.2.3). The capability is then made persistent so that further calls do not require contacting the binding service again.

Client Stub Generated

Example 4.3 shows the client stub generated for calling `putMsg` remotely. Because both string and integer are primitive data types in Napier88, the stub generator does not need to generate the packing and un-packing procedures for the types appearing in the procedure signature; it just uses pre-defined procedures for packing and un-packing scalar types.

The procedure `getServerStubName` accepts a value of type `Signature` representing the name and type of the procedure and returns a `ServerStubName` containing a field `host` (the server address) and a field `cap` (the capability to call this procedure at that server). The first time it is used, `getServerStubName` calls the remote procedure `import` at the binding service and caches the capability locally to avoid a further remote call (see section 4.2.3). The binding service address is returned by `getBinderName`. The procedure `clientStubMngr` implements the transport protocol at the client as described in section 4.5.1.

```
putMsg := proc( arg1: string -> int )
    begin

    ! check if capability has been imported, and import if not
    let sig := Signature( "putMsg", getType(any(putMsg)) )
    let ssn := getServerStubName( getBinderName(), sig )

    ! create the message to be sent to the server
    let sndmsg := Message( "", 1 )  ! initialise the message
    packInt( sndmsg, ssn(cap) )     ! pack the capability
    packString( sndmsg, arg1 )      ! pack the argument

    ! call remote procedure and wait for result
    let rcvmsg := clientStubMngr( ssn(host), sndmsg )

    ! un-pack and return the result
    let res := unpackInt(rcvmsg)
    res

    end
```

Example 4.3: Client stub generated

Generating Server Stubs

At the server side the application programmer interface is similar to the client side, although the implementation has three important differences.

1. The server stub generator now uses the values of the procedures in the environment passed as an argument.
2. The server stubs are stored by the stub generator automatically instead of being returned to the server program.
3. The programmer now calls a procedure that waits for incoming requests from the clients instead of calling the server stubs directly.

Example 4.4 shows how to create the message server. To generate the server stubs the programmer calls `makeServerStubs` with an environment containing *implemented procedures*. These procedures are not dummy procedures as in the client because these are the actual remote procedures that will be called by the server.

Although these implemented procedures need to be persistent in order to be called later by the server stubs, the environment with the implemented procedures need not be persistent

```
! create a map to store the messages
let mapOfMsgs := m_empty[int,string](eqInt,ltInt)

! create a variable for the number of stored messages
let nxtMsgId := 0

! create a new empty environment
let ServerProcEnv := environment()

! implement and insert putMsg into the environment
in ServerProcEnv
let putMsg := proc( msg:string -> int )
    begin
    nxtMsgId := nxtMsgId + 1
    m_isu_insert[int,string](mapOfMsgs,nxtMsgId,msg)
    nxtMsgId
    end

! implement and insert getMsg into the environment
in ServerProcEnv
let getMsg := proc( i:int -> string )
    m_find[int,string](mapOfMsgs,i)     ! assuming no errors

! construct the server stubs for all procedures
makeServerStubs( ServerProcEnv )
```

Example 4.4: Generating server stubs

itself as the process of generating the stubs will store these procedures by reachability from the server stubs (which in turn are made persistent by the stub generator).

Procedures beginning with "`m_`" create and manipulate maps [ABC+93, WWP+95] that provide a representation of mappings between any two types. A map is used, called `mapOfMsgs`, to associate a number (integer) with each message (string). Because `mapOfMsgs` is used by these procedures and these are persistent by reachability from the persistent stubs, the map itself persists.

There is another difference between generating client and server stubs. When a server stub is generated, the signature of the remote procedure and the server address are immediately exported to the binding service (see section 4.2.3). This is in contrast to the client stub, which waits for the first remote invocation to contact the binding service. This approach permits client stubs to be declared before their corresponding server stubs, the only obvious requirement being that these must exist for the remote procedure to succeed.

Server Stub Generated

The server stub for putMsg is somewhat trivial as shown in example 4.5. All server stubs are similar — even when the arguments have complex types — because the un-packing procedures for the arguments (and packing for the return value) either already exist or are previously generated and are just used by the stub. (The generation of packing and un-packing procedures is presented in section 4.4.3.)

A major difference between a server and a client stub is that instead of being called by the server program, the server stub is called by the transport protocol at the server side when a message for it arrives from a client (see section 4.5.2).

```
putMsgServerStub := proc( rcvmsg: Message -> Message )
    begin

    ! un-pack the argument
    let arg1 := unpackString(rcvmsg)

    ! call the intended procedure
    let res := putMsg( arg1 )

    ! pack and return the result
    let sndmsg := Message("",1)
    packInt( sndmsg, res )
    sndmsg

    end
```

Example 4.5: Server stub generated

4.3.2 Using the Stubs

After generating the stubs, the programmer then needs to use these stubs to perform a remote procedure call. Hamilton [Ham84] decided to extend the programming language with special syntax for calling the remote procedures, arguing that a language extension adds extra information for RPC control (such as server binding and error handling) and also emphasizes the semantic distinction between calls to local and remote procedures. Wai [Wai88] also extended the language in a manner similar to Hamilton for server binding, but without attempting to deal with errors at the language level because the objective was to support transparent distribution. Birrell and Nelson [BN84] provide such information in the form of extra arguments, a technique also used by Sun/RPC [Sun93b].

We chose *not* to extend Napier88 with extra syntax for calling remote procedures mainly for two reasons.

1. Extending the language would *force programmers to learn a new syntax* to use the mechanism, which can be a major drawback for its acceptance.

2. A language extension causes changes to the compiler that would *introduce difficulties for porting the mechanism* into future language releases. (For Napier88, an evolving research language, this is of paramount significance.)

However, if the syntax for calling remote procedures is identical to that used for local ones then the RPC mechanism lacks the semantic distinction argued as important by Hamilton. With identical syntax programmers should take great care when calling remote procedures, especially those problems related to the semantics of the arguments as described in section 4.4.1. (Subsequent versions of the RPC, described in chapters 5 and 6, use a different interface in order to return failures to the calling program and also to draw a programmer's attention to the different nature of remote calls.)

Using a Server Stub

After creating the server stubs the programmer simply calls `serverStubMngr` to start accepting incoming call requests from clients (see example 4.6).

```
! start accepting all call requests
serverStubMngr()
```

Example 4.6: Using server stubs

As described below in section 4.5.2, `serverStubMngr` represents the interface to the transport protocol at the server side. It is responsible for receiving incoming messages from the clients and calling the appropriate server stub based on the (capability embedded in the) received message.

Using a Client Stub

After calling `makeClientStubs` to create the client stubs as shown in example 4.2, the programmer can then use these stubs in a normal sequence of Napier88 statements. Example 4.7 shows how the client stubs generated can be used to test the message server. After this sequence of statements the server will now hold `"test msg"` indexed by the number 1 (meaning it is the first message to be stored in the database).

4.4 Parameter Semantics

With a syntax for calling remote procedures identical to that used for calling local procedures, one would expect the argument semantics of calls to local and remote procedures

```
! obtain the client stubs from the environment
use ClientProcEnv with
    putMsg:proc( string -> int);
    getMsg:proc( int -> string ) in
    begin

    ! store a message in the server
    let nbr := putMsg("test msg")

    ! retrieve that message from the server
    let msg := getMsg(nbr)

    end
```

Example 4.7: Using client stubs

to be the same. Unfortunately this is not true, as the physical separation of the application in different address spaces between the client and server programs imposes restricted semantics on argument passing and the range of transmittable types.

4.4.1 Passing Arguments by Value

A remote procedure is, by definition, executed in a different address space from where it is called. As suggested in section 3.3.6 and discussed in more detail in section 5.1, passing parameters by reference creates dependencies between stores that generally prevent the system from scaling to a wide-area or global network. This problem is exacerbated by persistence because these dependencies accumulate.

This is the reason why we chose to pass arguments by value, not only for scalars but also for complex values. Passing arguments by value means that the arguments are copied between the client and server programs. As these arguments may have references to other values, these must also be copied, and so on, until the whole transitive closure from the original arguments have been copied [HL82, Ham84, BJW87, Cra93].

When traversing the transitive closure of an argument, a value already copied may be found. This occurs for example when an argument A refers to two values B and C which share some other value D. The algorithm for deep-copying arguments preserves the semantics of this sharing for arguments to a remote procedure because sharing is the basis of many important data structures, e.g., doubly linked lists.

However, sharing is not preserved between arguments or between successive remote calls. For example, using the same example, if A is passed as an argument in two successive remote calls, then two distinct copies of A, B, C and D will be created in the server. Even if A is passed in different arguments *in the same call* multiple copies of A, B, C and D will

be created remotely. The same behaviour can be observed in many other RPC mechanisms. (We have implemented a partial solution to this duplication, described in chapter 6.)

4.4.2 Types Supported as Arguments

An RPC designer must attempt to support as many—and as rich a set of—argument types as possible. However, the remote procedure is executing in a different address space from the invoking procedure. This fact restricts the range of types supported by an RPC mechanism. Each restriction may result from some inherent difficulty or simply because its implementation and/or execution are too expensive.

We decided to support only a limited number of scalar types and constructors in Napier/RPC 1.0 and instead investigate other research issues, e.g., efficient type-checking using long-lived type sessions. (The next chapter describes Napier/RPC 2.2 which extends the range of supported types with richer constructor types such as procedures.)

Procedures for packing and un-packing scalar types are part of the RPC mechanism and they are used as appropriate within the stubs. We have implemented procedures for packing and un-packing the following scalar types: `int` (the set of all integers), `real` (the floating number data type), `bool` (with two possible values, `true` and `false`), `string` (all possible sequences of characters of any length), and `null` (with only one possible value called `nil`). The remaining graphical scalar types in Napier88 were not implemented.

The RPC generates a pack or an un-pack procedure on demand for each constructor type found in a procedure signature when creating the client or the server stubs. These procedures are compiled and put in the store indexed by type representation, in case one of these types appears again as an argument in another signature. This is possible only because Napier88 supports higher-order procedures and orthogonal persistence, and also because type representations are first-class values. The constructor types supported are: `structure` (labelled record type), `variant` (tagged discriminated union), `vector` (one dimensional array), and the recursive composition of these.

4.4.3 Packing Complex Types

It is not trivial to pack values of complex types because they may form a graph of references and sharing must be preserved.

Flattening Complex Types

In order to make the generation of packing procedures for complex types more tractable, the RPC first *flattens* any complex type into simpler type representations. The flattening process has two characteristics.

1. The new (flattened) type representation and the original type are structurally equivalent.

2. The new type representation has only one top-level type name and one constructor for each one of its sub-types (if any). This structure simplifies the manipulation of complex types and in particular the generation of packing and un-packing procedures.

For example, the result of the flattening process for the type `Signature` presented in examples 4.8 and 4.9 can be seen in example 4.10. (The type `Signature` itself is not important since it is used for exemplification purposes only. In addition, its use here should not be confused with its role as argument to procedures that implement the binding service.)

```
rec type TypeRep is structure (    ! record
    label:  int;
    misc:   int;
    name:   string;
    others: var;
    random: int )
& var is variant (      ! similar to union in C
    none:   null;
    one:    TypeRep;
    many:   *TypeRep;   ! '*' means vector
    unique: TypeRep )
```

Example 4.8: Type `TypeRep`

```
type Signature is structure (
    str:    string;
    rep:    TypeRep )
```

Example 4.9: Type `Signature`

This flattening process generates new type names, but as type equivalence in Napier88 is structural, both `Signature` in example 4.9 and `Type_Signature` in example 4.10 actually define the same type. The advantage is that we have a new type `Type_Signature` which can be manipulated independently of the original type (see below). In addition, there is no double constructor (like `many` in example 4.8) to complicate the type representation.

After being generated, the new flattened type representations are compiled using the *types compiler* available at run-time [KBC+94]. The result of the compilation is then stored in a special data structure called *declaration set* that is mainly used to group type definitions in the store. Later, a packing or un-packing procedure using this type can be compiled against the declaration set without the need to replicate its definition.

```
rec type Type__Signature is structure (
    rep:    Type__Signature__Str_rep;
    str:    string )
& Type__Signature__Str_rep is structure (
    label:  int;
    misc:   int;
    name:   string;
    others: Type__Signature__Str_rep__Str_others;
    random: int )
& Type__Signature__Str_rep__Str_others is variant (
    many:   Type__Signature__Str_rep__Str_others__Var_many;
    none:   null;
    one:    Type__Signature__Str_rep;
    unique: Type__Signature__Str_rep )
& Type__Signature__Str_rep__Str_others__Var_many is *
    Type__Signature__Str_rep
```

Example 4.10: Type Signature flattened

Generating Packing Procedures

The next step is to generate code to pack and un-pack these flattened complex types. We present in example 4.11 the generated code for `packType__Signature` that packs the top level `Type__Signature`. After being generated, this procedure is compiled and stored for later use. As can be observed in the source code, this procedure in turn calls lower-level packing procedures to pack the other generated types; these are not presented because they all obey the same basic structure.

The packing procedure generates a new `oid` for every value it packs, and stores the value together with its `oid` in the set `packedOfType__Signature`. (The `oid` is guaranteed not to repeat for 2^{32} packing operations.) Should the same value appear again *in the same argument of the same remote call* and the procedure just sends the number, it does not pack the same value twice. This is not only needed to preserve the sharing semantics (as described above in section 4.4.1) but also increases the packing efficiency and reduces transmission time.

In example 4.11 it can be observed that `oid`, even though it is represented by a number, is packed as a string. In fact, we decided to use strings as the basic data format for transmission over the network, instead of a binary data format such as those used by XDR or ASN.1 (see section 3.3.8). The reason for this decision was simplicity; this permitted concentration on other aspects of RPC like type-safety. (Later versions of Napier/RPC, described in the next two chapters, use the low-level Napier88 binary format.)

```
let packType__Signature := proc( sndmsg:Message;
                                 value:Type__Signature )
    begin

    ! find if ``value'' is in packedOfType__Signature
    let pos := ...

    ! if it is, then it was packed already
    if pos is found then

        ! value exists: pack only its oid
        packString( sndmsg, "@" ++ iformat(pos'found(oid)) )

    else

        ! new value to be packed
        begin

        ! create a new object id
        let oid := newoid()

        ! remembers ``oid'' and ``value'' as being packed already
        packedOfType__Signature := ...

        ! first packs ``oid'' then the value
        packString( sndmsg, "#" ++ iformat(oid) )
        packType__Signature__Str_rep( sndmsg, value(rep) )
        packString( sndmsg, value(str) )

        end

    end
```

Example 4.11: Procedure packType__Signature

Generating Un-packing Procedures

For every packing procedure on one side there is a corresponding un-packing procedure at the other side. For example, the procedure to un-pack a `Type__Signature` is called `unpackType__Signature` (see example 4.12). This procedure checks if the value has already being sent (signalled by the flag `@`) in which case it just looks for its value in a map called `unpackedOfType__Signature`. Otherwise the value is un-packed and stored in the map of values already received.

```
let unpackType__Signature := proc( rcvmsg:Message -> Type__Signature )
    begin

    ! get the number that identifies the value
    let str := unpackString(rcvmsg)
    let nbr := unpackInt(Message(str(2|length(str)-1),1))

    ! checks if value was sent before
    if str(1|1) = "@" then

        begin
        let value := ... ! get ''value'' from set with ''nbr''
        value
        end

    else

        begin
        let rep := unpackType__Signature__Str_rep(rcvmsg)
        let str := unpackString(rcvmsg)
        let t__Signature := Type__Signature( rep, str )
        ... ! put ''t__Signature'' in the map indexed by ''nbr''
        t__Signature;
        end

    end
```

Example 4.12: Procedure `unpackType__Signature`

4.5 Transport Protocol

The transport protocol is responsible for exchanging packets — byte arrays containing messages which in turn represent packed arguments and results — between the client and the server across the network. The transport protocol is built on top of sockets. Sockets [Sun93c] provide a convenient file-like interface to TCP/IP (see section 3.3.7) and are offered in Napier88 as a set of procedures found in the Standard Library [KBC+94].

4.5.1 Outgoing Calls

As we described in section 4.3.1, after packing the arguments into a message using the packing procedures, a client stub calls the *client stub manager* (`clientStubMngr`) in order to send the message to the server. The client stub manager implements the transport protocol at the client side and works as follows.

1. Accepts the server address and a message that contains the capability and the packed arguments for the remote procedure.
2. Uses the socket interface to send the message to that server address as a packet.
3. Blocks and waits for the result packet to arrive from the server.
4. When the result packet arrives, it just forwards it as a message to the calling stub.

Napier/RPC 1.0 creates a new socket connection for each remote call, as early experiments demonstrated that even this straightforward solution had acceptable overall performance. (The transport protocol has since then been optimised, see chapter 5.)

4.5.2 Incoming Calls

The *server stub manager* (`serverStubMngr`) is called once by the programmer to deal with all incoming packets, a process sometimes called *dispatching*. It works as follows.

1. Blocks waiting for a packet to arrive from a client.
2. When a packet arrives, it extracts the capability from it and identifies the procedure to be called.
3. It then transforms the rest of the packet into a message and calls the correct server stub using the message as its sole argument.
4. When the server stub returns, it transforms the result message into a packet and sends it to the client.

4.6 Summary

This chapter has described the design and implementation of an automatic type-safe persistent RPC mechanism built entirely in Napier88, a type-safe reflexive language. Type-safety is important since it guarantees that the signature of the remote procedure and that of the client stub are equivalent, thus eliminating a serious source of errors. Type-safety is especially relevant in persistent systems because the server may be used over long periods, comparable with the time over which software evolves. (An error may persist and become manifest at an arbitrary point in the future.) Type-safety is automatically guaranteed since no action from the application programmer is needed beyond standard declarations to make any procedures available via RPC.

The implementation follows that of Sun/RPC and generates client and server stubs which use packing and un-packing procedures. These are provided for base types and automatically generated on demand for complex types. The types supported are: integer, real,

string, bool and null; the constructor types structure, variant, vector; and the recursive composition of these. This is all achieved by code written in a high-level type-safe language without circumventing the type system at any point.

The implementation used many interesting features of Napier88 such as the type system, the callable compiler, higher-order procedures, and orthogonal persistence. These eliminate the need for an IDL, a separate language in which to write the signatures of the remote procedures. Instead, this RPC provides an *internal* stub generator that accepts signatures from within the language, generates new stubs, compiles and links them to the program, all at run-time.

This version of Napier/RPC was first described in a technical report [MdS95b] that includes its failure model. Later, it became part of Glasgow Libraries [CAL+94] and has since then been used by a number of application and system programmers. The applications built range from demonstrations (such as a card game) to persistent programming tools (such as the client/server Library Explorer described in chapter 7).

Chapter 5

Extending Object Migration

This chapter describes the two extreme models for migrating objects between stores — by reference and by copy — presenting their advantages but also the challenges they introduce. (In this thesis the word migration is an umbrella for all models of passing, transferring or transmitting data and code between address spaces. It does *not* mean movement with deletion of the original.)

We conclude that in a higher-order RPC, and especially in a persistent environment like ours, there is a clear need for compromises. Existing compromises are illustrated by giving examples of approaches that have been proposed to solve this problem.

We then present our model of migration by substitution. An object migrating by substitution does not actually migrate to the target store. Instead, the object is substituted by a surrogate. On arrival at the target, the surrogate is replaced by a local value equivalent to the original object.

The presentation includes the prototype implementation, an example of how to use substitution to migrate a procedure, and the applicability of substitution in persistent applications.

5.1 Migration by Reference

When an object migrates by reference from a source store to a target store, only a *proxy* for the object is sent to the target (see figure 5.1). That proxy may be an *object identifier* that already exists or it may be created for the purpose, but in either case it has to uniquely identify the object throughout the entire distributed application.

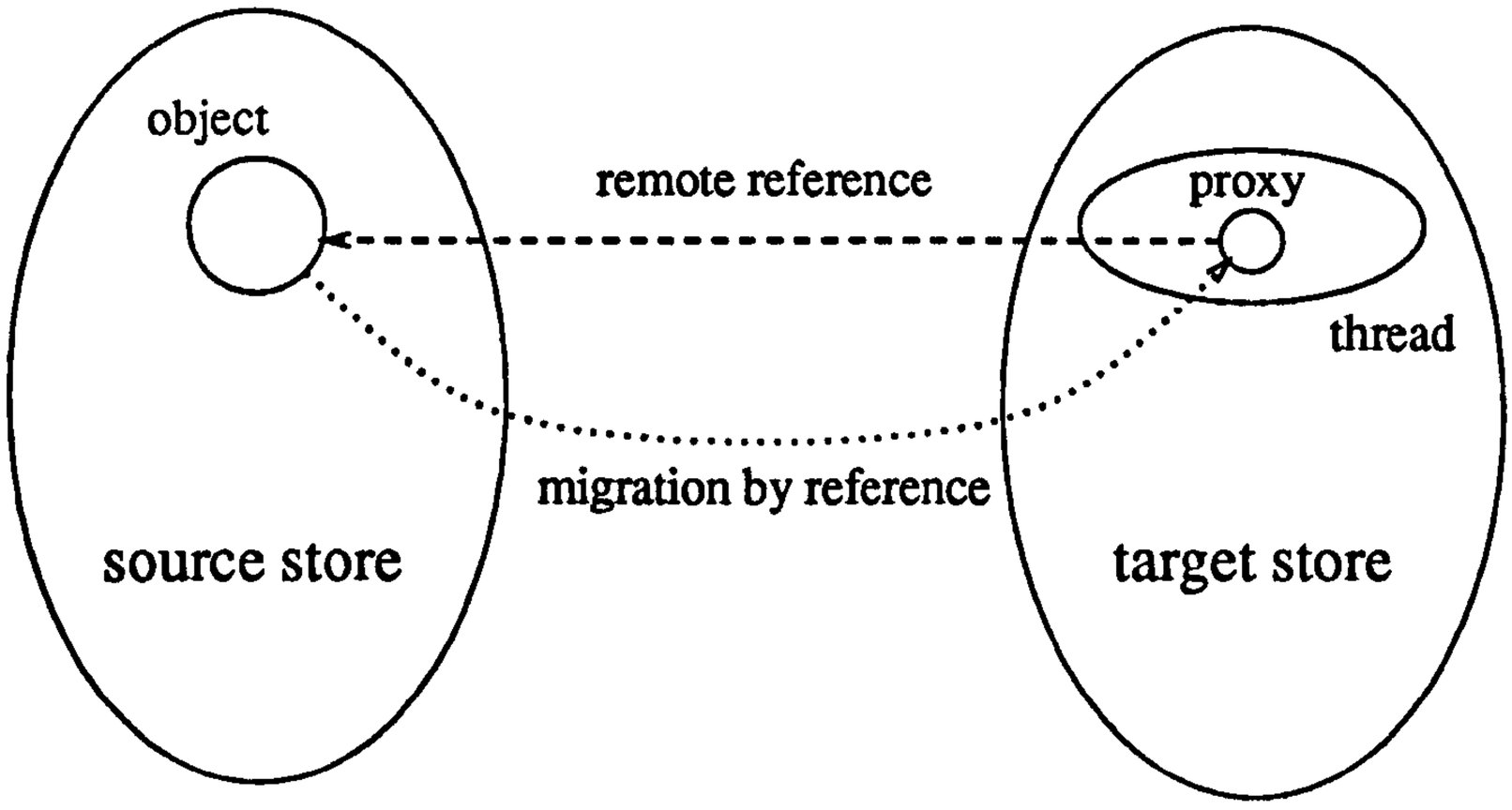

Figure 5.1: Migration by reference

5.1.1 Examples

Emerald [BHJ+87, JLHB88] is an object-based language and system for building distributed applications. Emerald can be said to offer *transparent distribution* since the application programmer does not need to deal directly with too many aspects of distribution. One of the main goals of Emerald is fine-grained object mobility, but it was not designed to support large-scale, persistent applications.

Emerald migrates objects by reference in all method invocations, local or remote, and the semantics are the same in both cases. However, there are two situations for which it is possible in Emerald to move the parameter objects to the remote program in order to increase performance.

1. The compiler may decide at compile-time to move an object at call time, for example small immutable objects such as integers or strings are (invisibly) copied.

2. An application programmer may decide to *suggest* copying an object to the remote program, based on knowledge about the application. This is achieved by using primitives in the language (see section 5.1.2 below).

Emerald is not persistent. In contrast, DPS-algol [Wai88] is a distributed version of the persistent programming language PS-algol [ABC+83]. DPS-algol also offers transparent distribution, but unlike Emerald it offers an RPC mechanism that gives programmers some control over the placement of computations.

If the object has a scalar type, the value of the object is copied to the program where the computation is executing (scalar types are immutable). Objects that are not scalar types are never duplicated in order to consistently support distribution transparency, including sharing semantics. Computations instead move from program to program collecting the values they need until scalar types are reached.

It was in DPS-algol that an accumulation of inter-store references was first observed to be a consequence of persistence. This is because computations visiting one store create new objects there which will never be able to leave that store. When DPS-algol was later used as a platform for building distributed applications, a failure in one store could prevent the entire application from working due to this inter-dependency between stores. These remote references also generated many small messages, delaying execution, saturating the network, and ultimately preventing the system from scaling (see section 2.2.1).

More recently, the research work by Kato and others on HiRPC [KOMM93, KKM94] has demonstrated that efficiency can be achieved in an RPC system based on call-by-reference. HiRPC adds a new language construct to C so that application programmers can choose between call-by-reference or call-by-move as in Emerald. Unlike Emerald, HiRPC also maintains a *cache* for frequently accessed remote data (coherency is automatically maintained). HiRPC also supports *call by variable-depth-copy* [KKM94] that can be fine-tuned to a certain level of the transitive closure.

Obliq [Car95a] is a distributed, higher-order, object-oriented language built on top of Network Objects [BNOW93]. Obliq is similar to Emerald in that only object references are transmitted in remote method invocations, but unlike Emerald objects are never automatically moved to another program as a parameter or result in a remote call. Like Emerald and DPS-algol, scalar values and other immutable values are simply (deep) copied. For example, procedure values (but not objects that represent mutable procedures) can migrate between programs.

These systems, especially HiRPC with its automatically maintained cache coherency, also introduce inter-store dependencies. However, since neither HiRPC nor Obliq handle persistence, these dependencies are typically not long-lived enough to be noticeable. Distributed persistent systems like DPS-algol are fundamentally different because they accumulate these problems and eventually stop working (see section 5.1.4).

5.1.2 Implementation

For computation to proceed, data and instructions have to be brought together in the same address space. This means that when the remote program needs to access the object, either:

1. the value of the object (or relevant parts of it) is then obtained; or
2. the thread of computation migrates from the target to the source program where the value resides.

In order to migrate the thread of computation, it is usually necessary to transfer objects which may include references to other objects. Hence the underlying support system must honour all exported references that may still be reached by all active programs. It also requires that inter-store references can be found as they form roots for garbage

collection and thus introduce the complexities of distributed garbage collection [PS95]. *In a distributed persistent application a remote access may occur an arbitrarily long time after the original object migration.* In short, migrating objects by reference creates the need for the network and the programs exporting these references to be running at all times (or at least to be able to be run).

Optimisation Strategies

Although migrating by reference is what should be ultimately perceived by the application programmer if RPC is to have a semantics close to that for local procedure calls, there are several implementation strategies that can be employed at the system level to approximate the same semantics.

Since data transfer is eventually necessary when migrating an object by reference, the implementation could instead duplicate the value of the object. If the object is mutable, then some *coherence protocol* is required to maintain the illusion that the object migrated by reference, e.g., [KSD+90]. The implementation of *identity tests* for duplicates and *distributed garbage collection* also becomes more complex [PS95].

Another alternative to implement the semantics of migrating objects by reference is to *move* the object to the server. This way, both object identity and shared access to the object are preserved because there is still only one copy of the object in the distributed application. But when moving the object *the source program no longer has local access to the object.* If it requests an operation on it, the object will "ping-pong" back and forth between these programs. Furthermore, all the references in the moved object become remote and when de-referenced will trigger further copying.

We have given examples of object movement in section 5.1.1 above. In Emerald there are language primitives that enable the programmer to give an indication as to how the system should execute the remote call. A hint, *call-by-move*, moves the object to the target program when the remote procedure is called (in the same network message as the call). A similar technique exists in HiRPC.

There is another primitive in Emerald, *call-by-visit*, a variation on *call-by-move* in which the parameter object moves to the target but returns to the source when the procedure finishes and returns the result. Note that neither mechanism replicates the object, so they are both still offering *call-by-reference* semantics without the need for a replication protocol.

5.1.3 Advantages

The main advantage of passing objects by reference is its *simplicity* because it offers the well-known local object semantics in a distributed environment. Passing by reference guarantees:

1. *Object sharing*—Its state is correctly shared by all its users at any one time; and
2. *Update semantics*—A distributed object has a well defined (in fact, unique) value.

5.1.4 Problems

Passing by reference gives the illusion that only a small amount of data (the object identifier) is shipped between the two address spaces. It may be the case that the remote reference is unused or passed back. However, sooner or later the computation may need to access the object.

There are basically two approaches that allow computations to proceed by collecting both computation and the objects referenced in the same address space.

1. the object is later migrated by copy and all local references to that object now become remote references (see figure 5.2); or
2. the computation, and a sufficient part of its context, moves (see figure 5.3).

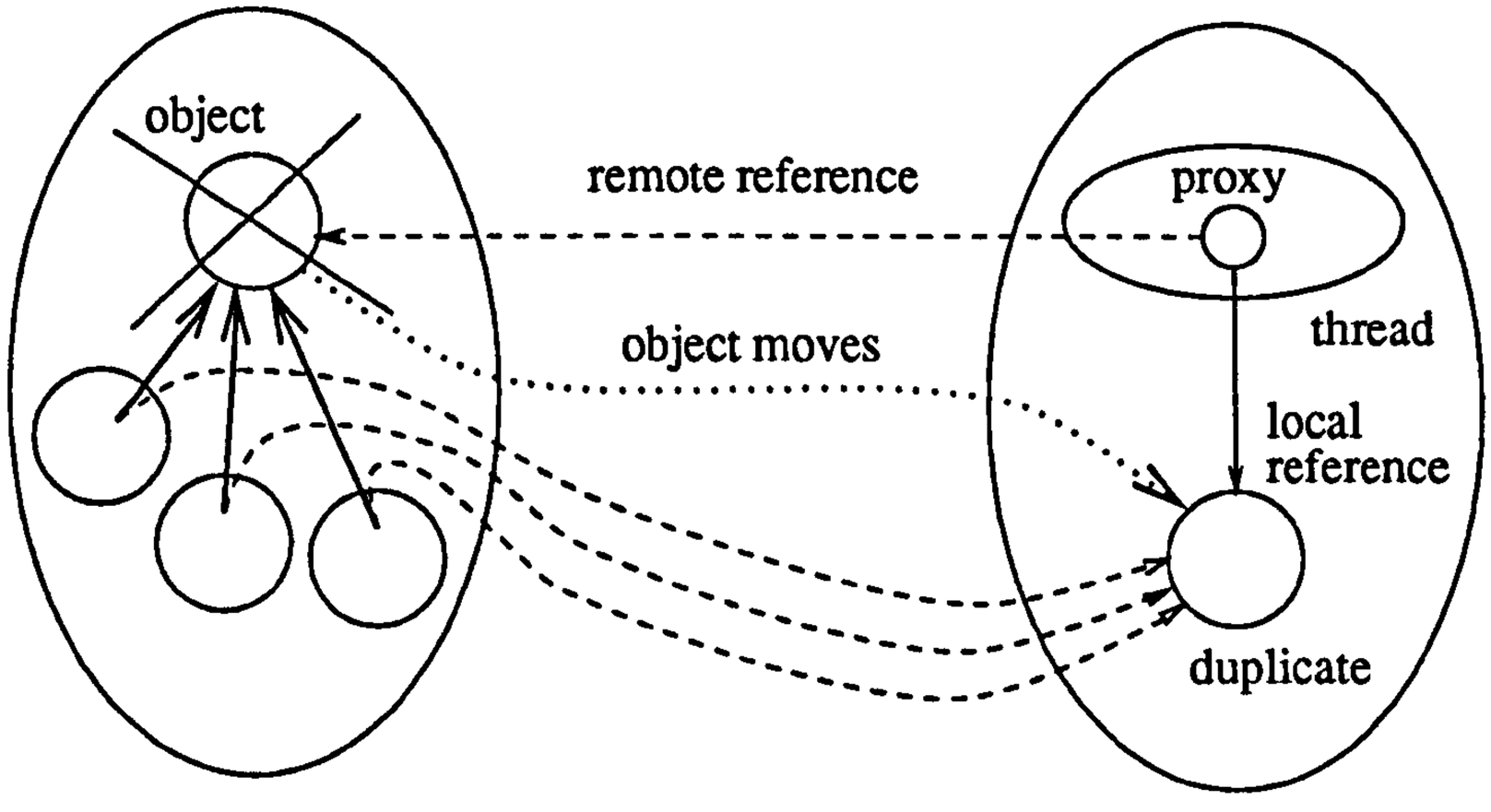

Figure 5.2: Migration by reference—the object moves

Migrating a small value (like an integer) across the network is typically 4 to 5 orders of magnitude slower than a local procedure call, thus migrating by reference can have a dramatic negative effect on the performance of the application [CKW96].

Passing by reference may or may not create more migrations across the network than sending the value right away, but less network traffic cannot be used as a justification for passing parameters by reference without stronger evidence. Furthermore, the resultant network traffic is decoupled from the call and this raises extra difficulty for maintaining semantics and failure handling (see below).

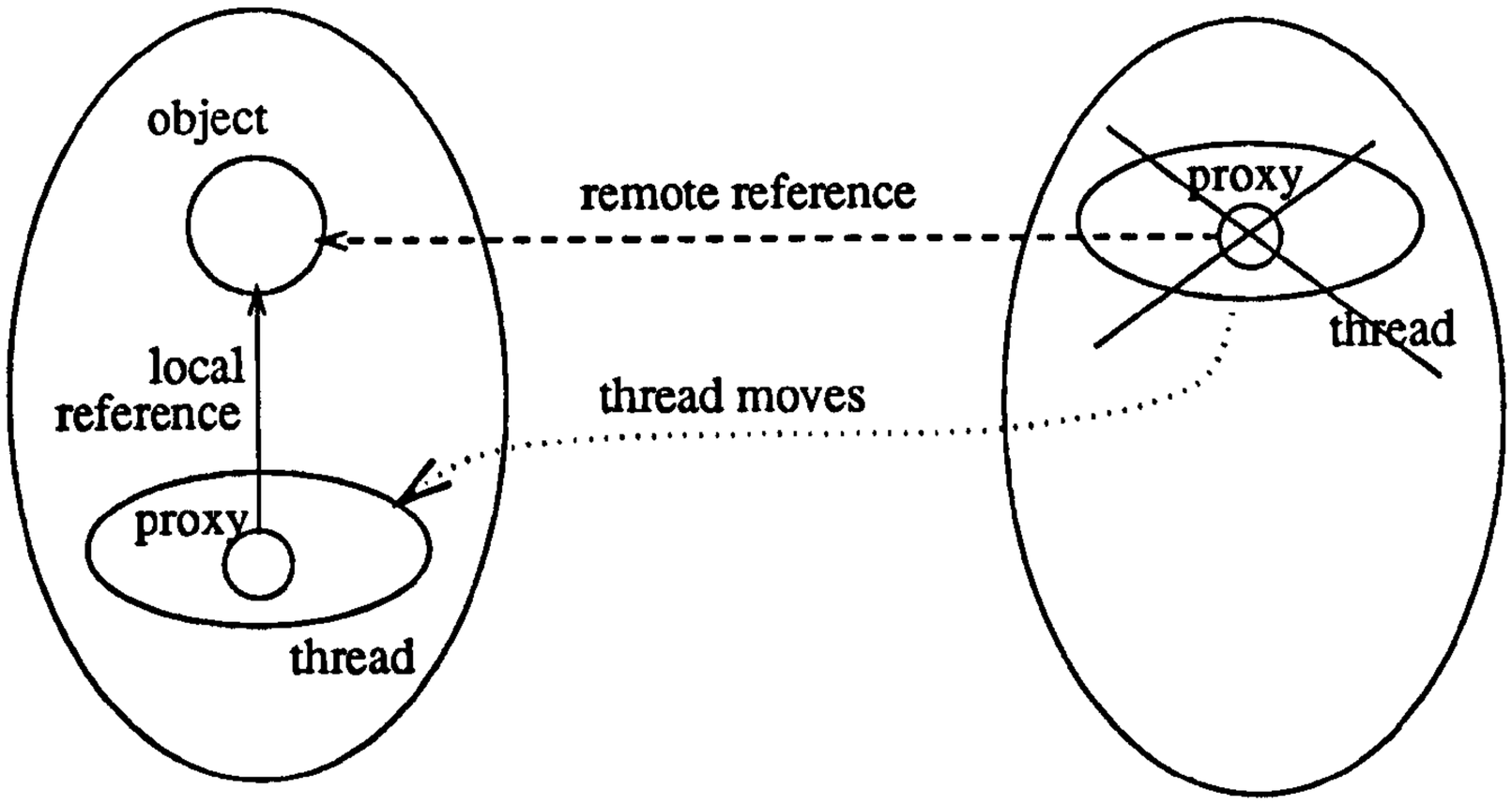

Figure 5.3: Migration by reference — the thread moves

The most important difficulty with passing parameters by reference is not network traffic, but that in the long-term these remote references create many dependencies between stores. Once shipped to another store, a reference may be assigned to local variables or potentially persistent object components, and dispatched to other stores. It is very likely that the objects that form a distributed application will become strongly inter-connected across the network, thus increasing the dependency between stores.

Dependencies can be acceptable in tightly-coupled distributed applications running on reliable local-area networks, but they prevent scaling of the application to levels where autonomy between stores is required. Autonomy is needed for many reasons, not least to cope with partial failures, and persistence implies that the application will eventually run long enough for partial failures to be significant.

The distributed system could try to hide all partial failures from the application. However, this is *unrealistic* because:

1. some failures may persist indefinitely (i.e., for any time longer than the people who will use the application are prepared to wait);

2. distributed garbage collection in an environment with partial failures presents well-known difficulties [PS95];

3. it is impossible to foresee all kinds of failures in advance; and

4. ultimately, the application will have to deal with unexpected conditions that can be modelled as failures.

Alternatively, the distributed system could recognize that remote references may break and provide exception mechanisms to deal with failures when accessing remote objects, i.e., passing failures to the application when it is unable to deal with them at the system

level. This also deals with failures where the user should be informed of the cause of delay or of (temporary) loss of data.

Although in some cases application programmers are better prepared to deal with failures than the distributed system due to their knowledge of the application, they should not be asked to deal with low-level failures as it unnecessarily complicates their work (e.g., network congestion). Further complications may arise because many of these remote references are created automatically by the system, and it is unfair to ask application programmers to deal with failures outside their domain.

5.1.5 Summary

Migrating by reference preserves object sharing, including update semantics, and thus provides to the programmer a simple distributed model. However, this simplicity brings a number of problems.

1. *Potential increase in network traffic* with unpredictable consequences on the application's performance.
2. *Dependencies between stores* that generate network traffic but also decrease availability and prevent the application from scaling to levels where autonomy between stores is required.
3. *Partial failures are amplified* creating difficulties for distributed algorithms (e.g., garbage collection) but they should not be passed up to application programmers.
4. *Semantics for orthogonal persistence, such as referential integrity, are no longer guaranteed* in a distributed environment with partial failures.

5.2 Migration by Copy

When an object migrates by copy, a replica of the object is created at the source and then shipped to the target store. Migrating by copy makes stores more autonomous because each store has now a local copy of the object and does not depend on the network or any other store to access its value.

However, the consequences of migrating by copy forces application programmers to keep the data being copied to a minimum by carefully designing the distributed application. They are also required to manage all the consistency and identity issues arising from multiple copies of the same object in the distributed application.

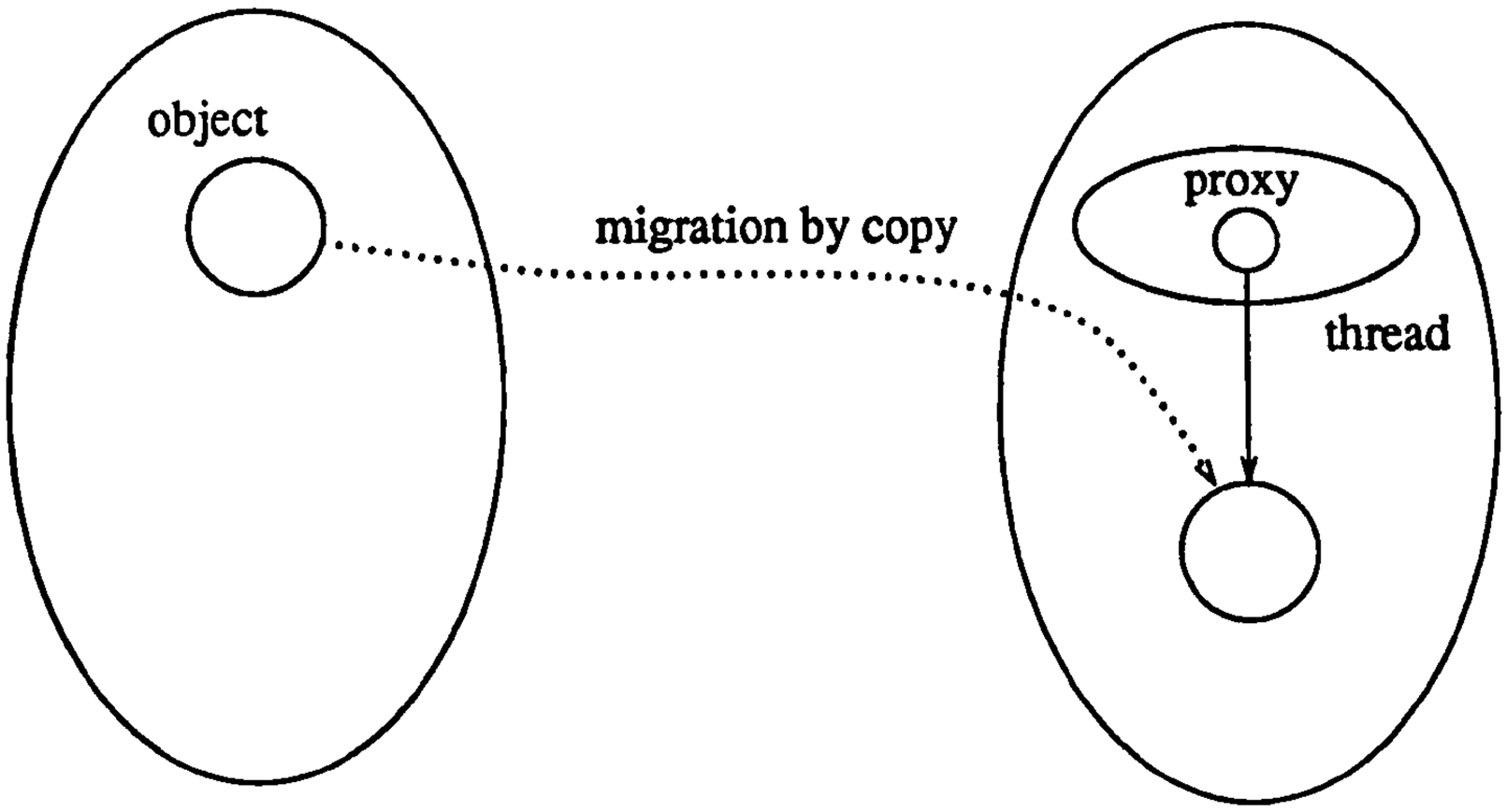

Figure 5.4: Migration by copy

5.2.1 Examples

Most RPC systems migrate parameter objects by copy, including the first RPC mechanism proposed by Birrell and Nelson [BN84], Hamilton's RPC designed for partial failures [Ham84], Argus [Lis88], the RPC products from OSF [OSF91] and Sun [Sun93b], Erlang RPC [Wik94], XEROX's heterogeneous ILU [JSS94] and Java's RMI [WRW96, RWW96]. (See section 5.1.1 for examples of RPC systems based on migration by reference.)

Parameter objects also migrate by copy in the first release of Napier/RPC, the type-safe persistent RPC mechanism described in chapter 4. Tycoon/RPC [MMS96, MMS95, Mat96] is a type-complete persistent RPC built for another persistent language called Tycoon [MMM93, MMS94]. While the first release of Napier/RPC generates its own marshalling stubs like most RPC systems, Tycoon/RPC uses an existing pair of procedures in Tycoon that marshal an object of any type to a byte array. Later releases of Napier/RPC use a similar approach.

5.2.2 Implementation

When an object is copied to a different program in a non-type-safe language like C++, pointers in the replica that refer to local objects become either:

- *dangling*—meaning that now they point to meaningless addresses; or
- *swizzled*—meaning that they are transformed into remote references.

In a modern type-safe language with referential integrity like Napier88, references are guaranteed to point to values of the correct type so *dangling pointers are not permitted.* On the other hand, *swizzling creates remote references* that are not guaranteed across an

unreliable network. Remote references suffer from a variety of problems as described in section 5.1.4.

The only realistic solution that maintains referential integrity and reasonable performance is to copy all parameter objects, and all objects reachable from these by transitive closure, preserving any shared and cyclic data structures [HL82, BN84].

These transitive closures can be quite large, but they are always limited by the address space of the source program. In a persistent language, however, the transitive closure of the parameters may include objects in the persistent store. Procedures that are first-class citizens may have very large closures—potentially as large as the entire store—because the "address space" in a persistent system is the persistent store.

Partial Copy

There are a number of implementation techniques that can ameliorate the problem of copying large transitive closures. For example, with *call-by-need* [TD94] the values are copied only as they are needed, not eagerly when the remote procedure is called. (Sometimes *call-by-need* is also referred to as *lazy-copy* or *fetch-on-demand*.)

Call-by-need tries to copy across the network only those values that are really necessary, but it may create many small network messages to fetch parts of the transitive closure. A variation of call-by-need is *call by pre-fetching*, in which more values than those that are actually necessary in the target program are copied in advance to accommodate eventual future needs. (This leads to the need for consistency and other problems, see section 5.2.4).

5.2.3 Advantages

If the value of the parameter object is copied to the target store, the source store is free to proceed autonomously until the target finishes its computation and is ready to send the result back. This is appropriate in large applications where: there is little control on the diverse stores that form the application; the source and the target are not guaranteed to be available all the time; the network can be slow and unreliable; and the computation runs for a long duration to balance these disadvantages.

More importantly than the physical characteristics of the distributed system are perhaps the semantic benefits of migrating by copy. With a local replica of the full transitive closure of an object, the computation in the target on this object is identical to the same computation on the original object in the source store. This semantic consistency includes failure modes.

5.2.4 Problems

The fundamental difficulties with migrating an object by copy are: the transitive closure that has to be copied across the network; and there is a potential for loss of consistency if a replica is created in the target store.

Transitive Closure

The transitive closure of an object creates three major problems: it may include large objects, objects that cannot be migrated or objects that already exist in the target store.

First, the time needed to pack, transmit and un-pack a transitive closure that includes one or more *large objects* may be significant. Even if the top-level object being copied is small, it is the size of its entire transitive closure that will restrict migration.

Second, a more fundamental problem occurs if the transitive closure of an object being migrated includes *objects that cannot migrate.* For example, when objects are fixed to a store for semantic reasons (e.g., there can be only one persistent root in each store) or implementation reasons (e.g., it may simply take too long to migrate the persistent root that includes the entire contents of the store). Other examples are: objects that only make sense locally (e.g., file descriptors to opened files, window handlers, and so on) and objects belonging to types for which there is not yet a migration implementation (e.g., threads in Napier/RPC, see section 8.3).

Finally, objects may already exist in the target store either because they are standard (e.g., a procedure to write a string on the screen) or because they have already been copied. Many objects can also be "standard" in a sub-set of stores by agreement between application programmers, for example in all stores that cooperate to form a distributed application. In any case these objects are sent again, not only decreasing migration performance but also using much needed store space.

Loss of Coherence

Loss of coherence happens because, when a mutable object is passed as a parameter to a remote procedure, *the new copy of the object that is created remotely is a different object* for all practical purposes. If either the copy or the original object is updated, inconsistent replicas will exist of what is still conceptually a single object.

Moreover, in a persistent system there may exist different programs written by different application programmers that cause repeated copies. If the same object migrates between the same stores again, in either direction, different copies of the same object will co-exist in the same store.

As an object may be copied repeatedly between stores, it may arrive at a store by several

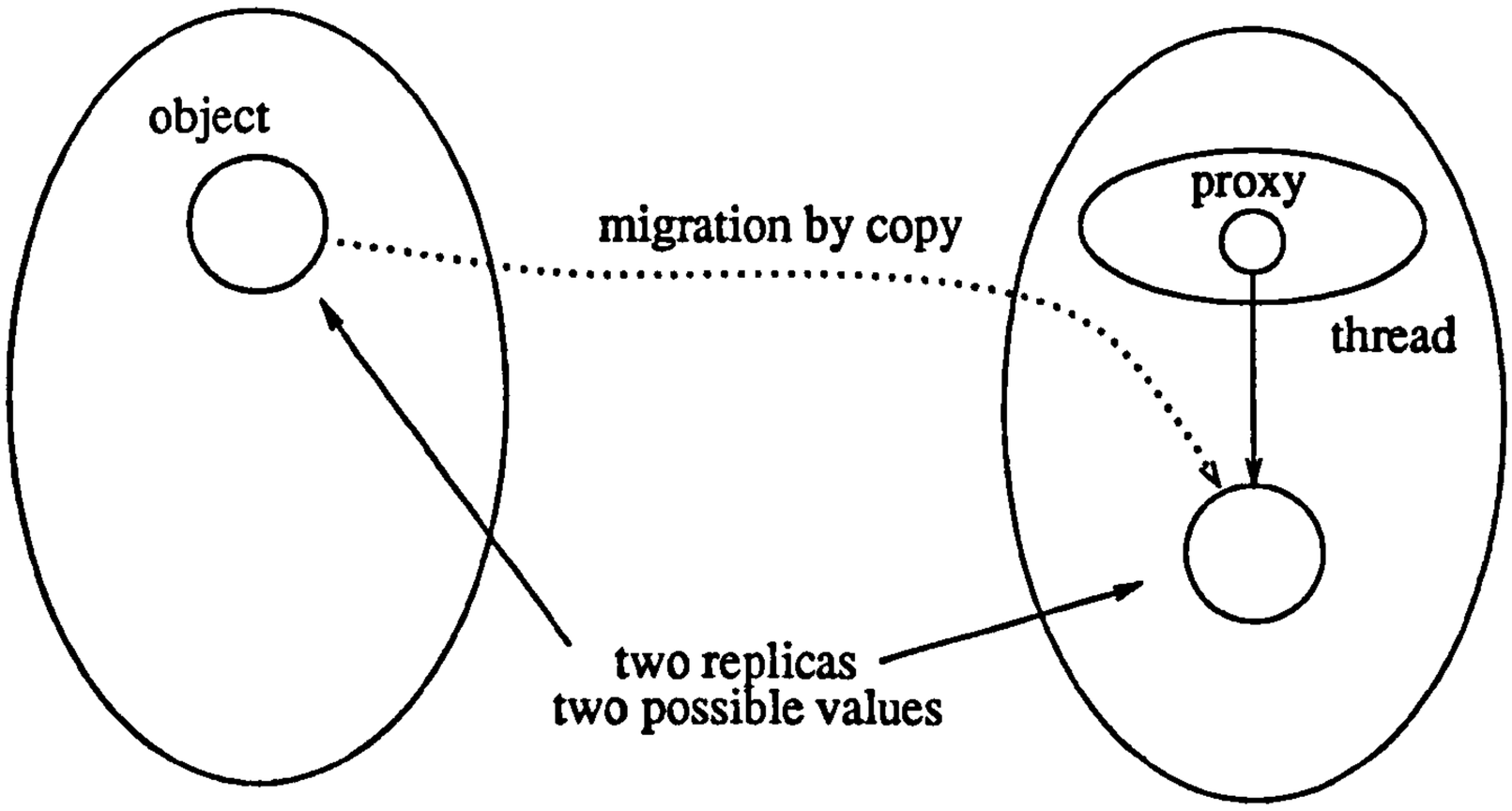

Figure 5.5: Loss of object sharing

different routes. This further increases the complexity and the cost of verifying identity and achieving coherent updates.

Problems that may be manifest with the top-level object may also occur for any object in the transitive closure of objects reachable from that object. Persistence amplifies these problems because transitive closures are potentially very large in a persistent store.

It may be simply too complicated for application programmers to deal with these inconsistencies, especially in a persistent environment. It may also be too expensive to attempt to eliminate these inconsistencies. The distributed system may help by providing certain primitives so that, for example, consistency can be requested explicitly by the application and this request will force an atomic refresh.

The system may also attempt to maintain total and permanent consistency between replicas offering to the application the illusion of only one object with one identity, e.g., by write-through cache with a primary copy [KSD+90]. This presents difficulties in large-scale applications since failures preventing the write-through may result in indefinite delays. Strict consistency creates dependencies between stores and has many of the same problems as migrating by reference (see 5.1.4 above).

Yet another possible answer to this problem could be *creating remote copies as immutable values*. Even though this changes the semantics of the copied objects, they will always reflect the value of the original objects at the time of copying because an immutable value cannot change. However, if the original object is updated, the copy now represents an old version of the object and its state is again inconsistent.

Finally, immutable values do not solve the loss of sharing when the object is copied back to the client. For example, when read-only parts of a document in a groupware application are collected in one store and distributed again by all contributors, a particular contributor will end up with many (possibly inconsistent) copies of the same part. Immutable values also introduce variations in the language's semantics dependent on the origin of a value

which contradicts the design philosophy guiding persistence — according to which there are no differences at the language level between local and external objects.

5.2.5 Summary

Migrating by copy creates duplicates and so permits each store to proceed independently from other stores. However, it also introduces a number of problems.

1. *Loss of object sharing* because many replicas representing the same object are (potentially) created in the target store, even if initially with the same value.

2. Not only the object but its transitive closure has to be copied to the target store, *decreasing migration performance* and extending the loss of sharing to the objects in the transitive closure.

3. Protocols to deal with multiple copies of the same object *simply re-introduce many of the same problems found with migrating by reference.*

4. *Persistence exacerbates all these problems* not only because transitive closures in a persistent system are typically large or very large (even the entire store) but also because the longer duration allows more replicas to accumulate.

5.3 The Need for a Compromise

Migrating objects by reference seems ideal because remote references are invisible — except for failures and performance — and maintain the existing (local) programming semantics, techniques and tools. However, migration by reference creates dependencies between stores that accumulate over time, especially in a persistent environment.

Operations on these inter-dependencies may increase network traffic and eventually saturate the network with small messages. They also amplify the effects of partial failures, which in turn destroy referential integrity. Furthermore, the complexity of garbage collection is one of the unsolved costs [PS95]. In short, remote references create difficulties for scaling the application beyond a few stores in a local-area network.

On the other hand, migrating objects by copy increases autonomy by duplicating their values in the target store. However, objects may have large transitive closures that have to be copied across the network, whereas only part of this closure will probably be used remotely. In the usual case where objects have up-datable parts, copying destroys the semantics of object sharing (common sub-structures are no longer common) and copies may diverge, creating inconsistencies.

For each technique *it is probably worthwhile to extend the resilience to failures* and hence to reduce the occasions on which an application programmer will need to be aware that the

distributed semantics differ from local semantics. However, reducing the number of failures and their longevity is possible only up to a certain point; after that point it becomes so expensive that other approaches would be preferred (such as limited inconsistency).

It is also recognized that *each mechanism can be made less susceptible to its drawbacks by applying clever implementation techniques*, such as cache coherence protocols in the case of copies and distributed reference management in the case of remote references. However, these techniques simply re-introduce the same or similar problems under another name. For example, cache coherence protocols need remote references and require a reliable network. These optimisations also make the implementation more complicated and thus dealing with partial failures even more difficult.

5.3.1 Existing Compromises

A number of compromises have been proposed between the extreme models of migrating objects by remote reference and deep copy. These compromises are typically based on passing objects by copy as a default, then passing some well-defined objects by reference in order to avoid copying too many objects across the network.

There are many examples of objects for which remote access may be better than copying their value to the target store.

1. *Large objects* for which only a small part of their value will be accessed in the target store, including databases. However, other large objects that are probably going to be accessed entirely should be copied, such as pictures. (How much of the object will be accessed depends on the application semantics and sometimes may vary dynamically.)

2. *Rapidly changing objects* with values that are updated frequently, such as a stock market value.

3. *Site-specific objects* such as specialised objects or objects that largely depend on their originating context. Examples include stores that take advantage of specialised hardware, large stores in mainframes, private or sensitive data or code, and so on.

Specified by the Application Programmer

Having decided that some objects will not be copied but instead accessed remotely, the next design issue is how the system decides which objects should not be copied. This decision is usually made by the *application programmer at compile-time.*

- The decision may be based on the type of the object, where some types are always passed by reference and all the remaining types are passed by copy. Examples include providing different object abstractions as in Argus [LS83, Lis84, Lis88], by inheritance

as in Network Objects [BNOW93] and implementations of CORBA [OMG95] such as Java/IDL [Sun96b].

- On the other hand, the distinction can be fine-tuned to the remote call itself. Examples include the *hints* (to move or visit, while still maintaining reference semantics) provided by Emerald [BHJ+87, JLHB88], the control over object location as found in Distributed Smalltalk [Ben87, Ben90], changing the marshalling code itself, as in Subcontract [HPM93], or annotating the methods in the schema with directives on how to pass parameter objects [Lop95, Lop96]

Done Automatically by the System

Instead of asking the application programmer to decide which types or objects should or should not be copied in a remote call, the system may attempt itself to provide this facility behind the scenes.

- The system may copy the first top-level, or a certain depth of the transitive closure, and provide remote references to those objects which are not copied [KOMM93, KKM94]. Only when an object which was not copied already is required, is it copied to the remote store. The system may remember how much of the transitive closure is typically accessed to optimise further migrations (although this has not been implemented to our knowledge).
- Instead of using the transitive closure, the system may attempt to copy automatically as needed but taking advantage of the characteristics of the underlying communication system by using a fixed size for the network message [TD94, THM+96].

Note that for these incremental or mixed-mode techniques, the implementation has to manage both inter-store references and detection of attempts to use remotely referenced objects.

5.3.2 Making Distribution Visible

All these techniques described above ameliorate the problem of copying too much of the transitive closure by achieving a trade-off between copying and remote references. However, all of them try to hide distribution when the application is executing and thus fail to cope with partial failures and different semantics (loss of object sharing, concurrency control, and so on).

We argue the techniques introduced above will not scale for large applications composed of a number of stores communicating over a relatively slow or unreliable network. For this kind of distributed application, the programmer *must* eventually be aware of distribution.

- When a store or a collection of stores fail permanently, then the application programmer is the only person who will know of a strategy for recovery or of an appropriate way of informing end-users.
- There are times when an application programmer will need reasonable intuitions about the cost and potential for failure of various operations.
- As federations are built, each application builder will need to have assurances about what data is copied where.
- Finally, some distributed applications will be so large or so geographically dispersed that automatic distributed identity, object management and coherency maintenance will be infeasible because of the combined effects of failures and communication costs.

Treating distributed computation in a manner different from local computation has been argued as necessary by others [WWWK94]. In the next section we present a new model of migrating objects that makes visible to the application programmer which objects migrate by copy and which do not migrate. Furthermore, we preclude the creation of remote references automatically at migration time; instead, the application programmer should use RPC (explicit remote calls) to access remote objects.

5.4 Migration by Substitution

In this section we propose a substitution model for migrating objects between autonomous stores. The model tries to achieve a compromise by incorporating the benefits, and avoiding the problems, of both migrating by reference and by copy.

5.4.1 Design

This new model of migration by substitution is based on a few assumptions.

1. *Only application programmers have enough knowledge about the application and its potential users to make the best decisions concerning object migration.* Thus we preclude automatic engineering optimisations done behind the scenes.
2. *Even low-level primitives can be used by application programmers* if they present a simple interface and are understandable without extensive knowledge about distribution (this is the case, for example, with CORBA). It is also important not to change the semantics between local and distributed objects without that change being explicit and clear to the programmers.
3. *Pairwise arrangements between stores, as opposed to global protocols, are acceptable* because many inter-relationships can be composed from these and failures inhibit

progress only when one of the stores in a pairwise agreement is unusable. ("Server" stores will have agreements with many other "client" stores, but these servers will form a minority within the entire store population.)

The primitives enable application programmers to work at the same level as remote calls, thus providing a consistent interface at the language level itself. This is in contrast to CORBA [OMG95] where a distinct "interface language" is used to identify which objects migrate by reference or by copy. The pairwise consistency arrangements are of limited duration based on *identity sessions*, analogous to the *type sessions* described in section 4.2.1.

The fundamental combined requirements for our new model of migrating by substitution may be summarised as follows.

1. *Prevent remote references across the network* that decrease store autonomy, create message traffic, amplify partial failures and have unpredictable consequences on performance.

2. *Avoid duplicating objects* if possible, for example those that already exist in the destination store.

3. *Provide a well-defined semantics for substitution* that has a simple interface and is easy to understand and use by the application programmer.

This composite requirement is not met by the basic parameter passing schemes that were summarised in sections 5.1 and 5.2, or to our knowledge by any of the variations that have been proposed by others described in section 5.3.1.

The Substitution Model

We propose a novel semantics for migrating objects between autonomous stores in a distributed application called *migration by substitution*. An object migrating by substitution does not really migrate itself, but instead only a *surrogate* identifying the object is sent to the target store. On arrival at the target the surrogate has to be replaced by an equivalent object with the same type—though not necessarily with the same value—as the original object.

Because the equivalent objects at the source and the target have different (local) identities and eventually different values (depending on the application semantics) the equivalence in our implementation is achieved by a *logical name* agreed between the source and the target stores. As usual we verify that the original and final objects have equivalent types.

We have described earlier in this chapter a variety of semantics for migrating objects. As figure 5.6 indicates, our model of migrating by substitution lies somewhere between pure deep copy and passing only a remote reference that points to the original object.

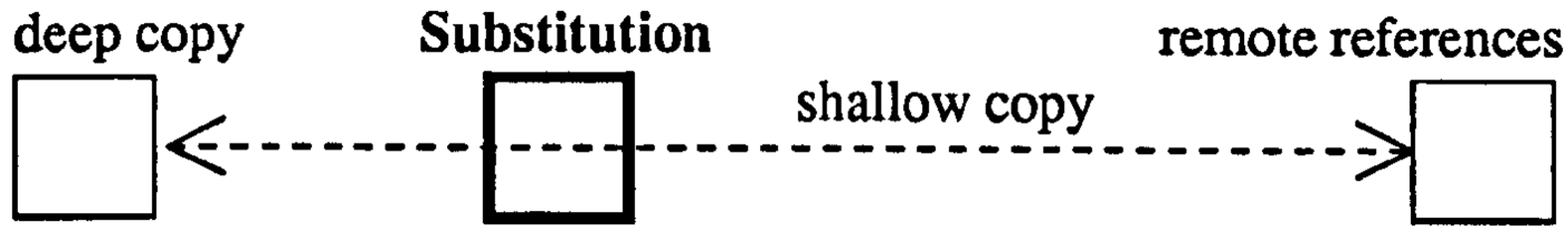

Figure 5.6: Migration by substitution

In this section we present the model and give an example of how to use it. The implementation will be described next in section 5.4.2. Related work will then be presented in section 5.4.3.

Partitioned Object Space

We need to introduce a few concepts in order to explain substitution. Local computation proceeds in a object space O. We then use a partitioning of the object space to establish the semantics regarding migration. The partition is currently into three *disjoint* sets, which are defined as follows.

1. *The Immutable Set* — Values of immutable objects are simply copied to the target store as the semantics of computation is unperturbed by their replication. We denote this set by the symbol I. This set includes all objects with scalar data types (integer, real, boolean, string, null and pixel) and constant objects with composite data types (see below).

2. *The Substituted Set* — Values of these objects are not copied; instead, they are substituted by a surrogate which is replaced by an equivalent object in the target store. For example, they may be members of the standard library for a persistent system [KBC+94]. This set, denoted by S, is *explicitly defined by application programmers.*

3. *The Copied Set* — The values of these objects are deep copied on transmission. They are denoted by C and are identified by set difference, that is, objects that are not in I or in S must be in C.

The partition above requires further clarification. An identifier in Napier88 is bound to a typed object denoting a value. The programmer specifies if the object is variable (the default) or constant. A constant object is similar to a variable object but it cannot be updated. (The Napier88 implementation detects any attempt to assign a new value to a constant object.)

In addition, an object can have either a scalar type or a composite type. Two scalar objects with the same type and same value are always identical, thus all scalar objects are constant and belong to the I set. In contrast, two composite objects with the same type and same value may have different identities. Thus composite objects belong to the C set if variable or the I set if constant.

The procedures and other objects which form the Standard Library [KBC+94] seem good candidates for an initial S partition. The implementation keeps a database maintaining an enumeration of this set S of substitutable objects (see section 5.4.2).

There is an interface that allows application programmers to add and remove items from this set. Although this set may be the same initially between all pairs of stores, we treat it as a pairwise agreement so that S may be changed between a sub-set of stores without global collaboration. Consistency between the two substitution tables in the source and target stores is not enforced, but migration will fail (an error code is returned back to the program) if a substitution has ocurred for a value that was not registered at the target.

Substitution is to be used as a primitive from which higher-level distributed protocols can be built, e.g., a set of mutually consistent substitution tables in a set of stores. A default store in Glasgow will be shipped with its substitution tables already set-up to include all values from both the Standard Library and the Glasgow Libraries. Then, application specific values can be added to the substitution tables by running a distributed set-up program based on RPC.

Algorithm for Substitution

Application programmers in the source and target stores need to agree and specify *the substituted set* by registering its objects by name in a pair of tables (see section 5.4.2). An initial set may be composed of all *standard objects*, that is, those guaranteed to exist in any store. For example, in Napier88 these standard objects are enumerated in the Napier88 Standard Library Reference Manual [KBC+94].

However, what is "standard" may depend on many factors. In Glasgow, for example, a typical application programmer may understand as "standard" all those objects that belong to the local Glasgow Libraries [WWP+95]. In addition, some programmers may agree between themselves what is "standard" for an application. This is the reason why, unlike other schemes presented in section 5.4.3, *our model of substitution permits objects to be added dynamically* (but safely) to the source and the target stores.

After the tables have been set-up with substitutable objects in both the source and target, the algorithm for passing a parameter by substitution works as follows.

1. If the parameter object has a *scalar type*, then it belongs to the immutable set and its value is copied to the target store.

2. If the parameter objects belongs to the *substitutable set*, then only its name is sent to the target store. (The name is guaranteed to be unique in both the source and target stores.) When the name is received in the target, it is used as a key to the local substitution table and the link to it replaced by a link to the equivalent local object.

3. If the object being migrated *does not belong to either the immutable or substitutable*

sets, then the entire transitive closure of its value is copied to the target store.

The algorithm is applied recursively to each reference to other objects embedded in an object being copied until a scalar type or a substitutable object is found.

5.4.2 Implementation

Migration by substitution was implemented by modifying a previous version of Napier/RPC described in chapter 4. Two modifications were necessary in order to support migration by substitution.

Firstly, the stubs that were automatically generated by the RPC system were replaced by general purpose procedures developed for supporting distribution in the Napier88 Standard Library (see section 3.4.1). This new version — which we usually identify as Napier/RPC 2, see section 1.3.1 — could now pass objects of any type as parameters or results. (The previous version, called Napier/RPC 1 and described in chapter 4, restricted the types permitted in remote calls.)

Without substitution, the challenges presented in section 5.2.4 invalidate migrating many objects with a complicated type, including many procedure values and instances of abstract data types. (This is especially true in the current implementation of Napier88 [CBC+90a]; however, some values of complex types in a strongly type-safe persistent language will always have large transitive closures.) The same problems plague the distribution support provided by the Napier88 Standard Library [Mun93, KBC+94] and restrict its usefulness. Overcoming this limitation was the main incentive for developing migration by substitution as described in this section.

The second change to the implementation is described below. In order to help the description, we give an example of a very simple procedure called `error` that migrates by copy to a target store. `error` makes use of another two procedures that migrate by substitution: `writeString` and `abort`. (A complete example describing how to use substitution from within the language will be presented in section 5.5.)

Step 1: Registration

As will be demonstrated in the next section 5.4.3 on related work, the fact that *the set of substitutable objects is defined at run-time* is a significant advantage of our substitution model when compared with similar approaches proposed by others. However, this flexibility has a drawback. How to make sure that names registered at different times at different stores refer to values of the same type ?

The substitutable objects `writeString` and `abort` are first registered by name in a *substitution table* (described below) at both the source and target stores. Their values and types are extracted by the mechanism and stored in the table alongside their names.

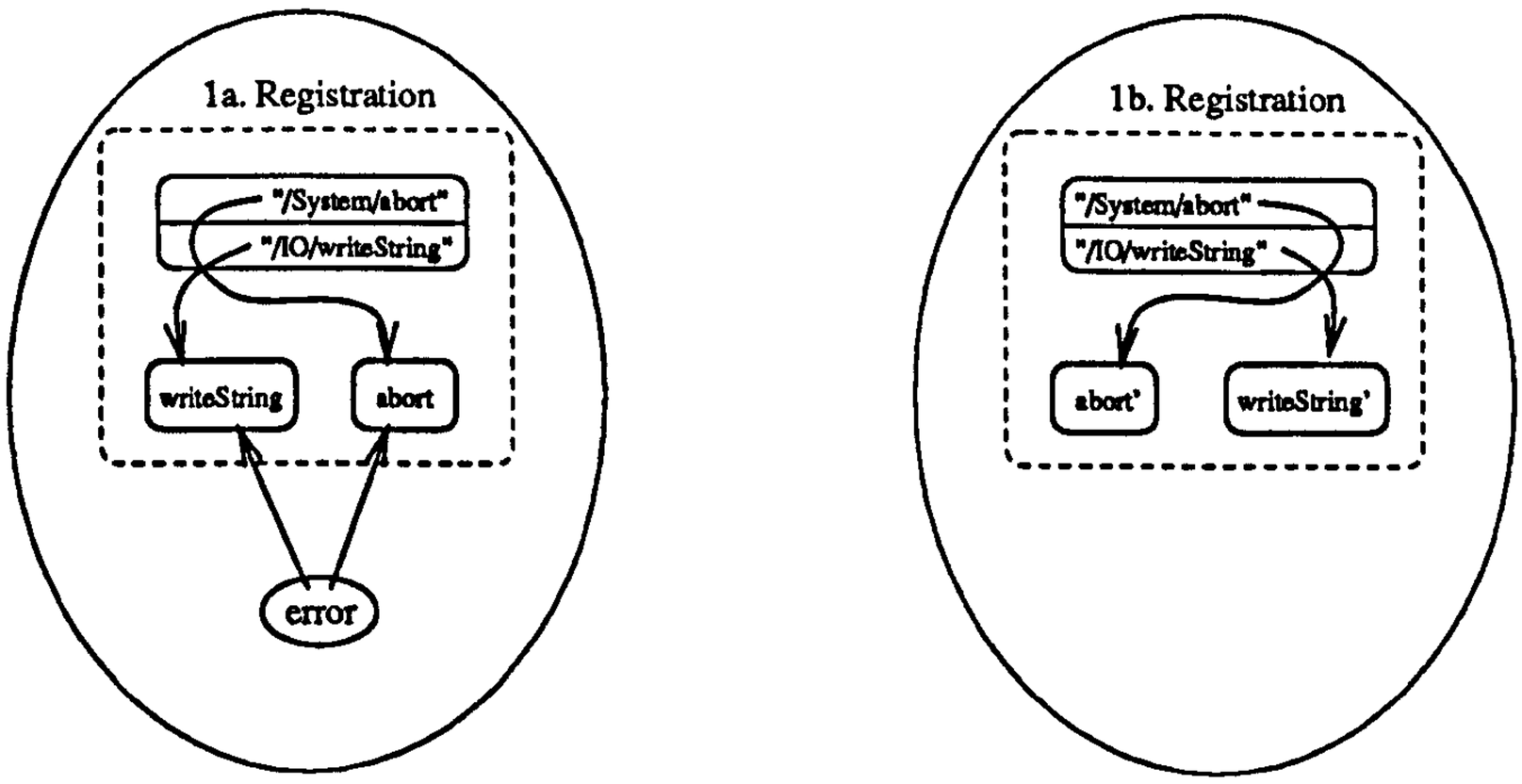

Figure 5.7: Step 1: registration

The substitution tables are just repositories for the names, values and types of substitutable objects. The only requirement for these is that they provide fast access. The extensive collection of Napier88 libraries include several implementations of efficient data structures. In particular, we have used the *maps package* [ABC+93, WWP+95] for implementing the tables at the target where access is by name.

However, there is a difficulty at the source that prevents an efficient implementation of substitution. Lookup in the substitution table is by value (not by name) and values can have any type (not always string). Although comparing two values for equality is simple and very efficient in Napier88 (essentially by pointer comparison), *there is no simple method for ordering two values with arbitrary types.* This means that a map from value to surrogate cannot be used. The hash table is another data structure providing fast access, but it is not trivial to generate a good hash algorithm for a value of any type. (We will come back to this problem in section 6.5.)

Step 2: Confirmation

The types of all substitutable objects registered in the target are *automatically* checked all at once against the source store as part of the first migration and they remain valid for all future migrations. (Alternatively, the application programmer at the source may request an *explicit confirmation* to check whether the names are valid prior to any migration.)

A variant of *type sessions* as described in section 4.2.1 was used in order to make our model of substitution strongly type-safe. Instead of checking signatures of remote procedures, type sessions now check the types of the substitutable values between the source and target stores. Type sessions are especially appropriate since types are agreed just once and the cost of type-checking is amortised in subsequent remote calls (migrations in this case).

It is up to application programmers in both stores to register the required objects for substitution. Because the substitution tables are persistent, separate (initialisation) pro-

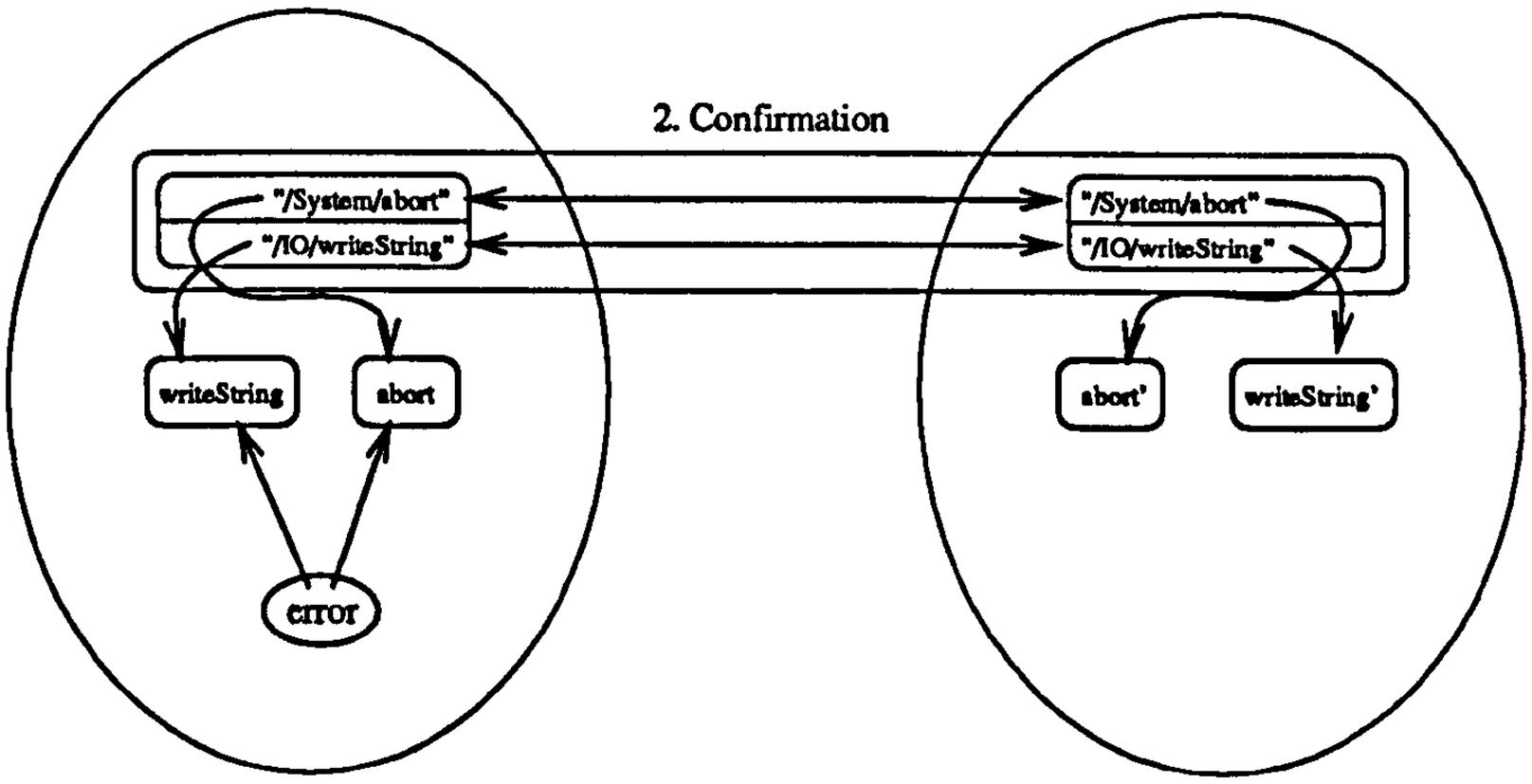

Figure 5.8: Step 2: confirmation

grams can set-up these tables before other (application) programs execute, thus removing the registration load from normal operation. (Indeed, standard utilities supply the entries for the Napier88 Standard Library [KBC+94] and Glasgow Libraries [WWP+95] in our experiments.)

Step 3: Cutting

When `error` is about to migrate, the transitive closure is traversed and the references to `writeString` and `abort` are *cut* and then *replaced* by references to their names. A flag is set-up in the implementation of the string to inform the target this is not a normal string but a surrogate for a substitutable object. (Special privileges are needed to implement this substitution, as the original type is temporarily replaced by a value of type `string`.) The migration algorithm does not look beyond the cut point since a string does not refer to any other objects.

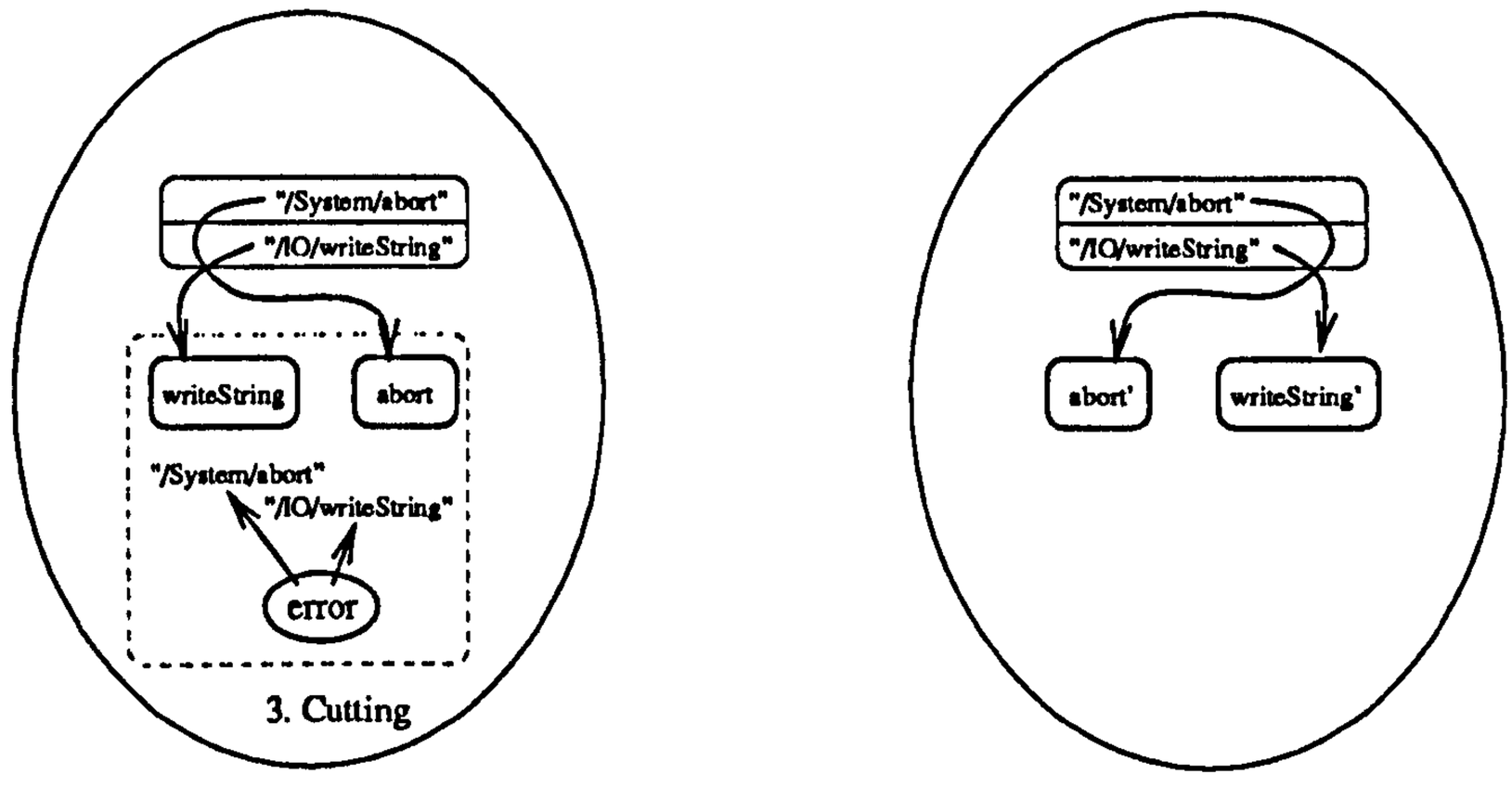

Figure 5.9: Step 3: cutting

Step 4: Transmission

The value of `error` is then marshalled into a byte array and this byte array simply transmitted to the target. On arrival at the target, the value is unmarshalled from the byte array into an equivalent data structure.

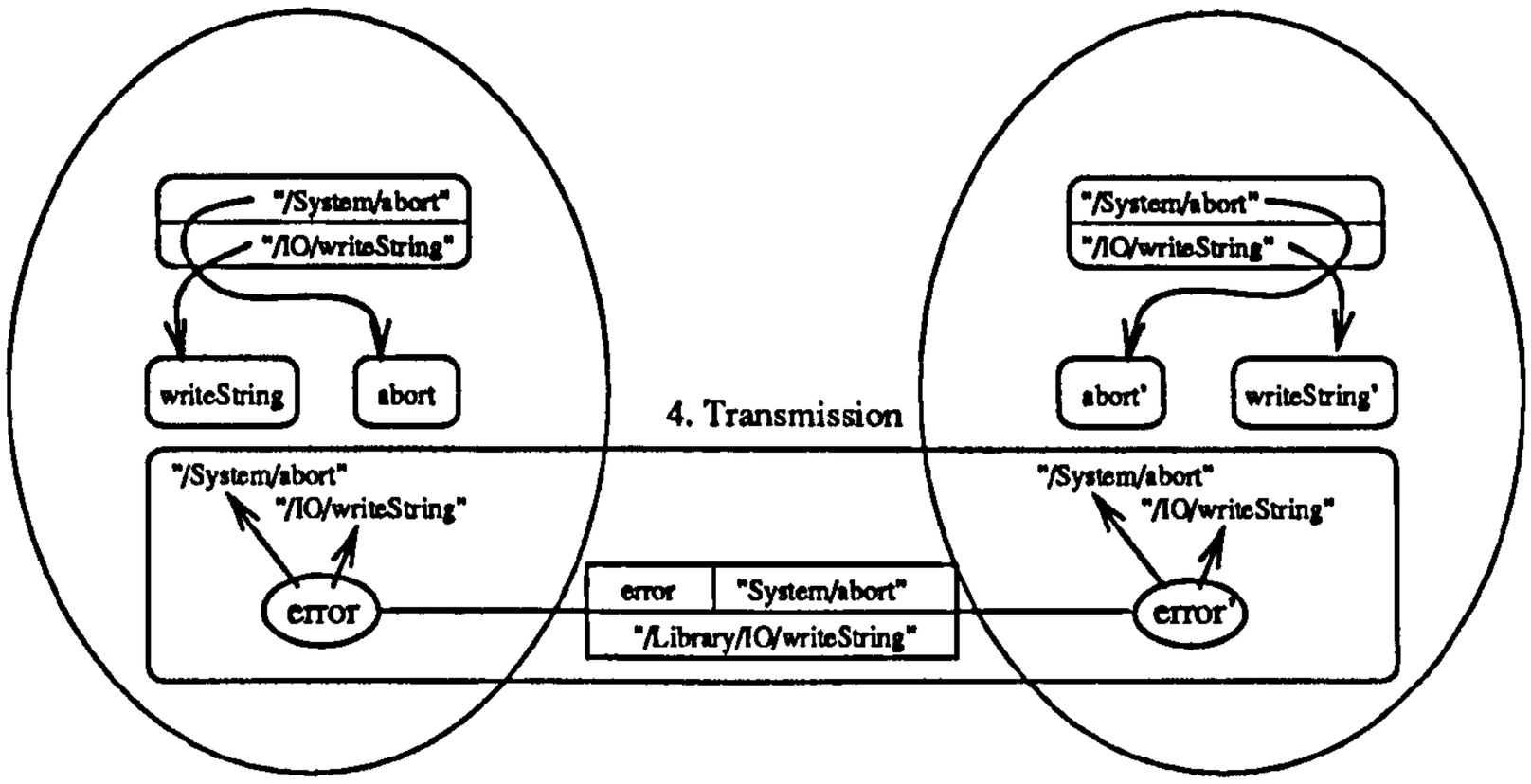

Figure 5.10: Step 4: transmission

Although *cutting* and *transmission* are presented as separate operations, in the actual implementation they are optimised by performing cutting while marshalling the object. The same goes for unmarshalling and re-binding. As a consequence, transmission now becomes just sending the byte array from one store to the other store. Since the byte array is just a sequence of bytes, the transmission can be made by any medium. In particular, we use "Napier88 sockets" (see section 3.4.1) for efficient migration, but there is no reason why it could not be written to a file and then be sent by e-mail, ftp, or floppy disk. (More on this in the next chapter.)

Step 5: Re-binding

The value is analysed recursively to check for the existence of surrogates. If one is found, a lookup by name is made in the substitution table and the reference to the surrogate replaced by a reference to the local, equivalent value of `abort`. Again, privileges are needed to circumvent the type-checking.

5.4.3 Related Work

In the previous section we presented a substitution model for migrating objects between stores. However, substitution in itself is not a new idea; other researchers (see below) have exploited this opportunity to avoid migrating standard objects that should exist—or are supposed to exist—in all stores that compose the distributed application.

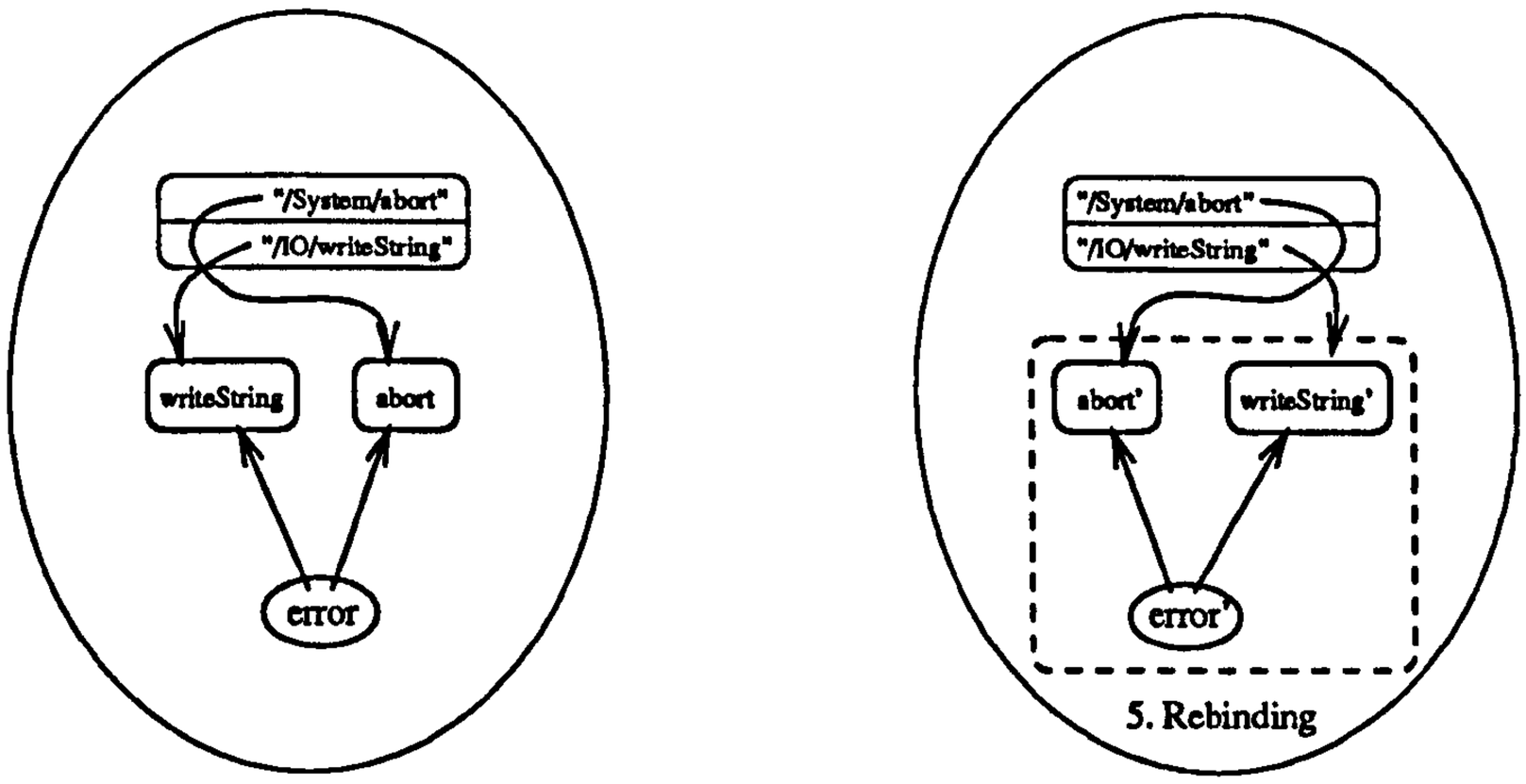

Figure 5.11: Step 5: re-binding

Ubiquitous Resources

Knabe proposes *ubiquitous resources* [Kna94, Kna95] that are identified *at compile-time* by the programmer using a keyword. In Tycoon/RPC *ubiquitous values* [MMS96] can also be defined at compile-time. These provide a restricted form of substitution where values originating from the same compilation can be substituted.

Other programming languages are even more restrictive. Java [AG96] supports *applets* (classes compiled to byte code) that can migrate between programs on the Internet. *Applets* dynamically bind to local versions of the Java class library, but attempts to bind to other classes would fail.

Migratory applications [Car95b, BC95a, BC95b] in Obliq [Car95a] are more interesting because they are represented by closures and thus may contain *live data*. (Java *applets* cannot, creating difficulties for the use of Java for distributed computing based on mobile agents [MdSA96a, ADJ+96].) Obliq agents were designed for graphical environments and for interacting with end-users; Obliq (like Java) uses the local (standard) library for interfacing with the user.

Configurable RPC

Modern RPC systems introduce greater flexibility for specialising migration. For example, Sun's Spring [HPM93] lets programmers customise marshalling and unmarshalling to suit their own requirements. It is possible to implement something similar to our substitution mechanism by replacing these operations with custom-built marshalling code, but this task may require extensive knowledge from the programmer about distribution and how RPC systems work.

Network Objects [BNOW93] also allows programmers to specify *custom procedures* for pickling particular data types (which could, in principle, be used to implement the substi-

tution mechanism we describe) but details of how this can be achieved are not presented in the paper.

A technique designed explicitly for manipulating bindings in a type-safe persistent environment is Octopus [FD93, FD94]. Octopus allows type-unsafe operations only in meta-space, but type-safety must be obtained before objects are dropped into the value space again. Although it was originally designed with distribution in mind, it does not explicitly support distribution abstractions. Both the meta-space concept and lack of distribution support may make it unsuitable for direct use by application programmers. We are unsure of its present implementation status.

Comparison with Substitution

The novelty of our model of substitution lies in the combination of the following characteristics.

- *Simplicity*—Because substitutable objects are defined *by name*, the model is more easily understood and used by typical application programmers. Names for substitutable objects can be matched against names in the application, such as paths from the persistent root for library procedures. Error messages are meaningful in case of migration failure.
- *Flexibility*—Registration is made at *run-time* so it can be delayed until just before migrating an object. This permits the programmer to extend or reduce the set of substitutable objects dynamically if required by the application. For example, an application for remote installation of libraries may define as "standard" the components which have just been installed.
- *Type-safety*—Simplicity and flexibility are not achieved by relaxing the safety nets provided by persistent systems because we use *type sessions* to confirm that equivalent objects in the source and target stores have the same type.

Similar substitution models have been proposed before, but they all fail to support the combined characteristics of simplicity, flexibility and type-safety supported by our model of substitution.

5.5 Higher-order Migration

Migrating procedures between stores is especially challenging in the current implementation of Napier88 because *most procedure values include the entire store in their transitive closure.* In this section we describe why plain migration by copy does not solve the problem with large transitive closures and how substitution can be used to provide at least a partial solution.

5.5.1 Example of Migrating a Procedure

Example 5.1 presents the source code of a procedure called *error* that an imaginary application needs to migrate to another store. The procedure simply prints a message of type **string** on the screen and aborts the current execution. As with many procedures in Napier88, **error** binds dynamically to other procedures to make use of existing code. In this example, both procedures **writeString** and **abort** belong to the Napier88 Standard Library [KBC+94].

```
use theStore() with Library: env in
use Library with IO,System: env in
use IO with writeString: proc(string) in
use System with abort: proc() in
    begin

    let error := proc( msg:string )
        begin
        writeString( "ERROR: [ " ++ msg ++ " ]'n" )
        abort()
        end

    end
```

Example 5.1: Source code of **error**

There are two major issues in migrating **error** to another store: the data format for transmission and the migration semantics. These issues will be discussed below.

Data Format for Transmission

In Napier/RPC, procedure values are transmitted in their hyper-program format [Kir93]. Hyper-programming is a novel format for storing programs made possible by persistence because only in a persistent system the source code can be compiled and put into the store together with the data and other procedures that are bound to the program. An *hyper-program* represents a Napier88 program in the store and is composed of source code and bindings from that source code to the objects needed by the program.

It is important to note that there is nothing in hyper-programs specific to Napier88. Hyper-programs are just an intermediate format of source code representation between text and byte code. The same techniques of substitution are applicable to byte code in Napier88 and other languages, and in particular there is no need to change the compiler.

Figure 5.12 shows the hyper-program of **error** presented in example 5.1. (The picture was taken using the hyper-programming environment based on the Napier88 proprietary window manager, see section 3.2.4.) From this alternative representation of the procedure

source code, we can perceive how `error` is bound to the two other procedures `writeString` and `abort`. (The *L*, abbreviation for *location*, means there is a dynamic binding from the boxed procedure names to their values.)

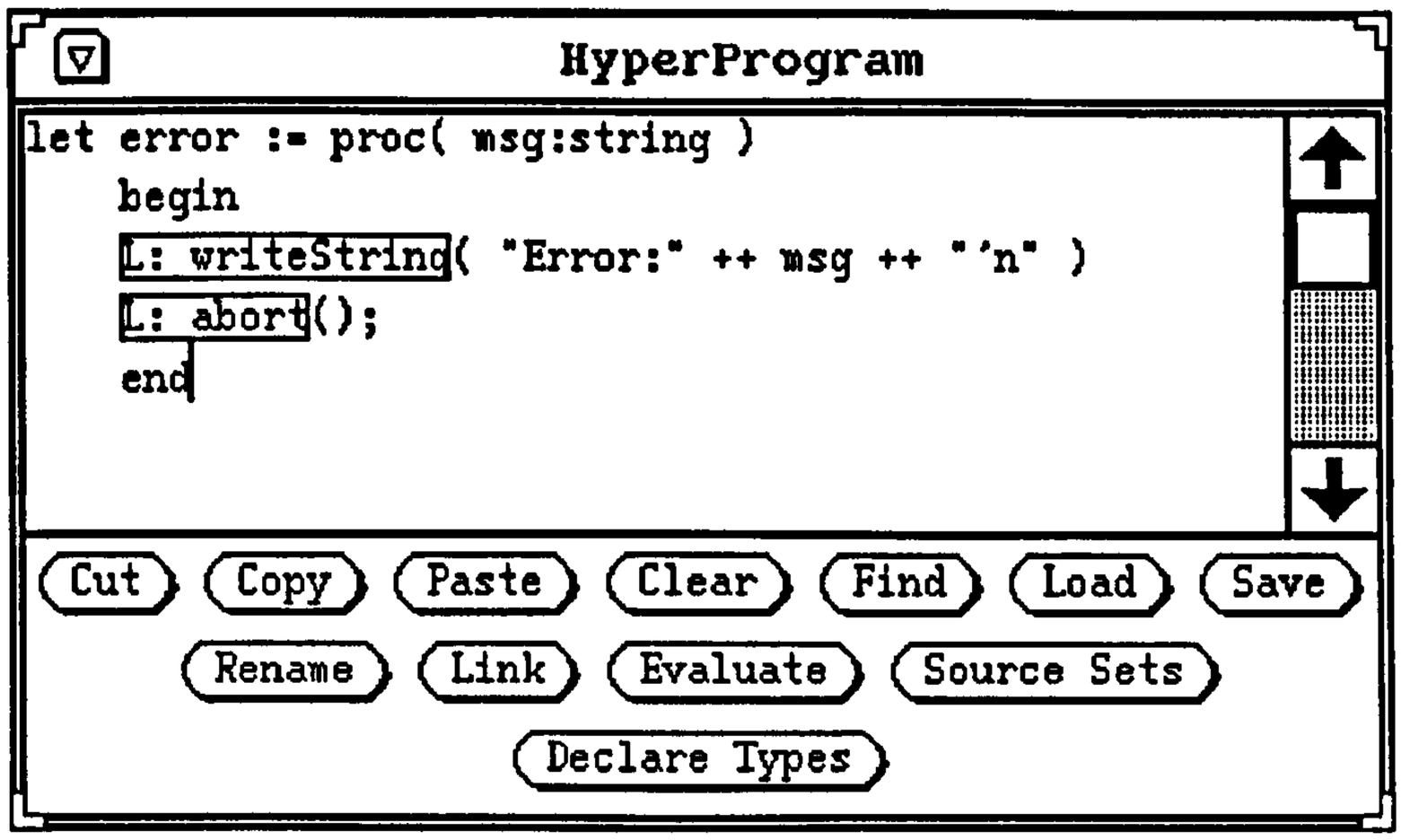

Figure 5.12: Hyper-program of `error`

We chose hyper-programming because it facilitates program manipulation (see below). However, passing procedure values between stores in hyper-program format has two major drawbacks.

- *Performance*—Procedure values have a reference to their hyper-programs in Napier88 so access to them in the source store is efficient. On the other hand, the inverse process (from hyper-program to the procedure value in the target store) requires a compilation which is potentially much more CPU intensive than re-building the value directly. (Preliminary performance measurements are presented in section 7.2.2.)
- *Lack of protection*—Hyper-programs contain textual representations of the original code and as such represent a potential security hole during transmission. (But not locally, since they are compiled before being returned to the client program.)

There are other possible formats for transmitting procedure values between stores apart from hyper-program, namely as byte code like Java *applets* [AG96]. For the current version of Napier/RPC we opted for hyper-program instead of byte code for the following reasons.

- *Simplicity*—Hyper-programs are easier to manipulate than byte code because they are represented by first-class values in Napier88.
- *Safety*—It is dangerous to manipulate byte code directly because it works below the safety net provided by the Napier88 type system.
- *Portability*—Hyper-programs are more easily portable between machine architectures and implementations of Napier88, e.g., the current abstract machine [CBC+90a] and the new PamCase [CCM95].

Migrate by Copy

A first attempt to migrate **error** by copy to the target store is presented in example 5.2. (The procedure **migrate** accepts an **any**, the infinite union of all types, to permit a value with an arbitrary type to be transmitted.)

```
! try to copy error, writeString and abort to the target
migrate( target, any(error) )
```

Example 5.2: Migrating **error** by copy

This migration attempt by copy will recursively copy **error**, then **writeString**, then everything needed by **writeString**, then **abort**, then everything needed by **abort**. This solution is not recommended for three reasons.

- *Difficult*—Both **writeString** and **abort** are complicated procedures, partially implemented within the abstract machine. Even though it is conceptually possible, there is no easy way to copy these procedures since parts of their values are "hard-wired" into the system itself.
- *Not needed*—There are already local copies of **writeString** and **abort** in the target store since they both belong to the Napier88 Standard Library [KBC+94].
- *Not efficient*—Migration by copy duplicates these standard procedures in the target store every time a procedure (or any other value) referring to them migrates. As a result, CPU time, network bandwidth and store space will be wasted (especially since this duplication is not needed).

Migrate by Copy with Substitution

The recommended solution is to migrate **error** by copy but both **writeString** and **abort** by substitution.

Example 5.3 shows how the application programmer can register at run-time the *names* for **writeString** and **abort** at the source store. The registration makes these procedures ready to migrate by substitution to the store called **target**. (A similar registration process has to occur at that store.)

The migration algorithm starts by copying the value of **error**. Since **error** is a procedure, the reference to its value is substituted by its hyper-program representation. (A different flag from normal substitution is set-up to signal the original value was substituted by an hyper-program, not a surrogate.)

The hyper-program is then copied recursively to the target. When the algorithm encounters a link to the value of **writeString**, it checks to see if the procedure has been declared as

```
! declare these procedures as substitutable
! the values will be accessed by following the path
substitute( target, "/IO/writeString" )
substitute( target, "/System/abort" )

! send the value to the target store
migrate( target, any(error) )
```

Example 5.3: Migrating `error` using substitution

substitutable. If it is, then the value is replaced by the surrogate "/IO/writeString" and the marshalling process continues.

During the respective unmarshalling process at the target, the two surrogates are replaced by the local equivalent values of `writeString` and `abort` and the hyper-program of `error` is built. (The migration fails if these standard procedures have not been declared substitutable in the target, see section 5.4.2.) Finally, the hyper-program is compiled to produce the procedure value of `error`.

5.5.2 Applicability of Substitution

In order to understand the usefulness and practical worth of substitution, we have analysed the Munros application. Munros is the name of a persistent application written in Napier88 to manage information about Scottish mountains higher than 3,000 feet — called Munros — including when they have been climbed and by whom.

Dag Sjøberg measured the Munros application with his set of tools for analysing persistent applications based on a thesaurus [Sjø93, SCWA94]. There are in total 381 values declared by dynamically binding a variable name to a location in a *environment* (see section 3.2.1). Of these, only 26 — or 6.8% — are values created by the Munros application itself [Sjø96]. The remaining 355 values — or 93.2% — are either part of the Napier88 Standard Library or Glasgow Libraries.

Although these figures might be considered to support evidence that the Munros application makes extensive use of libraries, the numbers by themselves do not necessarily prove this [Phi96].

1. Munros could have been written in a single module to avoid dynamic binding or for any other reason.

   This is not the case. The Munros application was developed following the methodology proposed in [Sjø93]. As a result, the entire application is composed of 20 files and the average number of lines per file is 80.

2. Munros could have been written in a way to reduce dynamic binding between compo-

nents of the application. For example, grouping related procedures in structures or abstract data types is a well-known higher-order programming technique to replace a number of individual values by a single top-level one.

This is also not the case. The 26 declarations actually refer to 11 values: 5 procedures, 2 variants, 2 structures, 1 vector, and 1 picture. The variants and vector contain sets of data. The picture is a map of Scotland. One structure contains 12 procedures, but the Munros application only makes use of 1 of these. The other is an instance of the WIN window manager, but the Munros application only makes use of 4 procedures. On the other hand, many of the values in the Standard Library also represent sets of procedures (e.g, an instance of WIN, the window manager).

The Munros application is an example of an end-user persistent application. The measurements presented above suggest that, in general, end-user persistent applications make extensive use of libraries. This in turn means that substitution has potentially wide applicability, provided libraries and applications are both designed to take advantage of software re-use.

5.6 Summary

This chapter has described two extreme models of parameter passing—by reference and by copy—including examples of distributed systems based on them, their design issues, advantages and challenges. We concluded that neither of these models is ideal for building large, distributed, persistent applications: migrating an object by reference creates unpredictable network traffic and depends on the availability of the network and other stores, whereas migrating by copy duplicates values generating semantic and performance problems. These problems are amplified in a persistent environment where links between objects are long-lived and transitive closures typically large.

We then argued for a compromise between these two models in an attempt to combine their strengths and reduce their problems. Existing compromises are presented, but all of them are either based on application programmers making decisions at compile-time or just engineering optimisations at run-time. Furthermore, they also reduce copying by creating remote references, an approach suitable for small-scale reliable environments, but not desirable in larger applications.

We then proposed a new model of migrating objects which does not create remote references. In our model, called migration by substitution, programmers can decide at run-time which objects should be copied and which objects should not. Objects that are not copied are substituted by a surrogate which is sent to the target store. At the target the surrogate is replaced by a local equivalent version of the object. This technology does not depend on changes to the compiler and thus applies to other higher-order persistent languages.

Chapter 6

Persistent Spaces

In this chapter we propose a new IPC model called *persistent spaces*. A new model is justified because sharing objects between autonomous persistent stores is not conveniently supported for a certain class of applications by any of the existing IPC models (see section 6.1). Persistent spaces extend migration by copy and they do not require migration of objects by reference or substitution; they are proposed in addition to — and thus can be used in conjunction with — these existing IPC models.

The chapter includes the motivation for persistent spaces, the design and semantics of the operations supported by them, the application programmer interface, an implementation in a persistent language and a comparison with related work. (An *example application* and *performance measurements* of persistent spaces are presented in the next chapter, and *future work* in chapter 8.)

6.1 Motivation

Persistent spaces are useful for sharing the values of complex persistent objects between a number of autonomous stores. The main difficulties with existing models arise because of: *partial failures* on computers, programs and the network; and the *transitive closures* of objects that can reach significant parts of the store.

Large transitive closures are a consequence of the *complexity of the objects* we choose to represent, and orthogonal persistence. For example, procedures have typically very large transitive closures because they are bound to other procedures in the store, and these to other procedures, and so on. As a consequence, large graphs can be found in Napier88, as in any other persistent language with an equivalent, higher-order type system.

6.1.1 Target Applications

This chapter concentrates on *sharing persistent objects* for a sub-set of all distributed persistent applications. We suspect this sub-set, based on the assumptions below, includes many practical applications for which the existing IPC models are not appropriate.

1. The number of object servers is much smaller than the number of clients of these objects. Servers usually have far more computing resources (disk space, processing power, access to broadband networks) and human resources (for building and maintaining code) than clients.

2. The number of objects that migrate from one server to a particular client is relatively small compared with the total number of objects in the distributed application—although not necessarily small in absolute terms. This is because it is more cost effective to access remotely (at the server) very large objects or objects that are seldom used by clients.

3. Objects made available by servers to be consumed by clients have often reached some kind of "stable" state (for example, they are the result of other computations). In addition, clients may not always require the most up-to-date version of all objects. However, they do need a facility to check if objects have changed and to access the latest version if required.

4. Network use may be slow, unreliable and expensive compared with computation and data movement within computers. This is especially true when considering the network available for the client (e.g., a notebook connected *via* wireless modem). There is no indication these differentials will be substantially reduced in the foreseeable future.

Examples of this sub-set of distributed persistent application include: a diary shared by a research group; document databases (e.g., for technical or sales support); development of large applications in a team; and distribution of software. Currently these applications represent only a small niche in the software market, but they will grow in number and importance with the rise of the Internet and so-called *intranets* (private networks built with Internet technology).

On the other hand, we recognize these assumptions will not hold for many applications—for example, those targeted by distributed object-oriented databases [Che93]. For these applications, some other distribution model that may be more expensive or more difficult to use than our proposed model is required.

For example, the basic RPC mechanism can always be used to fetch the most up-to-date version of an important and rapidly changing, relatively simple, object—the canonical example being a stock market value. Remote references can be used when remote access is required only sporadically or the object cannot be copied for semantic or implementation reasons. More likely, a combination of IPC models will be used in most distributed persistent applications.

6.1.2 The Case for a New IPC Model

This section motivates the need for a new IPC model. The argument will be based on:

- *the various costs of achieving the desired behaviour* in terms of programmer time (learning and using the model to build the application) and system costs (CPU usage, network load, and store space) to access the values of large objects from multiple stores; and
- *the adequacy of each IPC model in realistic conditions*, with particular reference to the response to partial failures by the application program.

Remote References

A remote reference to the only copy of an object is the obvious solution to sharing the value of an object between a number of stores. However, in section 5.1.4 we have described how remote references may degrade performance and introduce uncertainty in the application due to partial failures.

Performance is limited by communication costs every time the local application reads or writes the value of a remote object. The methods described in section 5.3.1 can optimise performance, but only up to a certain scale.

The main difficulty with remote references is that they introduce *uncertainty*: since remote references have an interface similar to local references, network load and partial failures are referred to the application program out of context. Application programmers cannot make use of their knowledge to try alternative procedures, cache values, or at least present the end-user with a reason why computation is not proceeding.

Local Copies

Another solution to the problem of sharing an object is by duplicating the value of the remote object in the local store. Access to the object only requires a single remote connection (to copy the value) and autonomy between stores is guaranteed after that operation. However, local copies also have a number of problems.

Firstly, assuming consistency is required, the local and remote stores cannot operate on the object for long because the copies will eventually become inconsistent. (On the other hand, if consistency is *not* required—or at least not required all the time—then copies may be the solution, for example, if the object is immutable.)

Secondly, if only a small part of the object value is updated, then only that part needs to be transmitted. However, most IPC mechanisms will just (re-)transmit the entire transitive

closure of the object. This can be acceptable when objects are relatively small, but becomes very inefficient for large transitive closures as happens in a persistent environment.

There is no straightforward solution to avoid re-transmitting the entire transitive closure of the object every time part of its value is updated. For example, using an "open RPC" like Subcontract [HPM93] is not an approach that most application programmers will be eager to use since it requires understanding of low-level RPC issues. Instead, programmers will probably try to reduce the functionality of the application to limit the complexity (and so the size) of the objects being shared.

A third problem is that many copies of an object will accumulate in the same store, and in a persistent environment these copies survive program executions. Apart from being a poor use of store space, there is a semantic problem because updating one copy will not update the others. A knowledgeable programmer can always write code so that a new copy will replace an existing copy, but this effort distracts programmers from their applications.

Migration by substitution

Substitution, as proposed in the previous chapter, can also be used to share an object between a number of stores. Substitution has two advantages compared with the models described above: it does not require remote references between stores and it does not create multiple copies of the same object in the local store (of course, only for those objects that are substitutable).

One limitation of substitution is that it only works if the object being shared has a name associated with its value. (This is a design decision; objects without a name are also potentially substitutable but they are not easily recognized by the average persistent programmer.) Many of the objects being shared can be protected by mechanisms in the language, e.g., procedures that hide a shared data structure. Substitution would require programmers to expose these objects and destroy their protection.

The other limitation is that the value of the object needs to be highly stable because substitution only supports sharing if both objects (in the source and target stores) have the same value. If one object changes, then programmers have to transmit its new value to the other store using some other mechanism (with all the problems found with local copies, see above) and update the substitution tables to refer to this new value. Although substitution provides a partial solution to sharing, it still requires another mechanism and some programming effort to coordinate them.

Replicated Objects

Replication is another method to share a remote object by duplicating the value of the object locally (that is, a replica in each store where the object is used) and maintaining the value of the replica consistent with the original. This guarantee of consistency avoids some

of the problems found with basic duplication. It can be used together with substitution to provide a complete solution to the problem of sharing objects between stores. There are, however, two main problems with replication.

Firstly, replication protocols that maintain strict consistency between replicas are based on remote references and suffer from the same fundamental problems—or even worse, depending on the number of replicas, the update pattern, and the replication protocol itself. Furthermore, strict consistency protocols can also be very expensive in CPU time and network load.

The second problem is that strict consistency is *not* useful for applications in which some stores are simply *not* interested in accessing the latest value of the object most of the time. Many applications either have this requirement already or can be designed in a manner to cope with certain levels of inconsistency for semantic, performance, management, or any other reason (see section 6.1). They are not ready to pay the price for something they do not need.

Conclusion

This section motivated the need for a new IPC model that supports sharing of large persistent objects between autonomous stores. We are particularly interested in a class of applications for which some large, complex, persistent objects must be shared but consistency is not required everytime and everywhere. These applications are not prepared to pay the price for strict consistency if application programmers can decide when consistency should be achieved.

There are a number of other issues. Control over remote operations is important, in particular to write programs that are aware and can respond to network and server failures. Just duplicating objects cannot be the answer because copies would quickly become inconsistent. On the other hand, if programmers are to control sharing of persistent objects effectively, then the new IPC model still has to be simple to understand and use.

6.2 Model of Persistent Spaces

Persistent spaces are based on a very simple, one-to-many, weakly-consistent replication protocol. We call them *persistent* for two reasons: they were designed to build distributed persistent applications and an implementation for them is more natural in a persistent language.

Persistent spaces can be distinguished from existing IPC models in one crucial aspect: while most of the functionality required for building the class of applications described in section 6.1.1 is *maintained* by the persistent space, all remote operations are explicitly *invoked and controlled* by the application programmer.

This separation permits the programmer to build the application in such a way that *more decisions about network costs and partial failures can be taken depending on the application semantics.* This is in contrast with actions being taken automatically by the distributed system based on an "average" application. On the other hand, persistent spaces still present an interface simple enough that most application programmers can use the mechanism.

6.2.1 Overview

A persistent space is a repository of objects that is *published* by a server and *subscribed* to by any number of clients.

1. Objects are *put* into a persistent space by an application program executing at the server.

2. Any client that has subscribed to the persistent space can, at any time, *fetch* all objects put into the space—the only remote operation. (In contrast with strict replication protocols, the model does not give any automatic guarantee of consistency.)

3. Later on, the client can *build* the objects locally, thus independently of network and server availability.

An example with one server and three clients is depicted in figure 6.1. The server publishes the persistent space, which is subscribed by all clients. Subscribers 2 and 3 have fetched the contents of the space, but client 3 is the only one that has re-built the objects locally.

A space is built incrementally, transmitted incrementally, and guaranteed to rebuild its objects incrementally at any subscribing client. The price to pay is that *all subscribers receive all objects put into a space* when they fetch that space (although after the first fetch they only receive object updates or new objects, see below).

On the other hand, *marshalling at the server is done only once for each object* and the cost will be amortized over all clients. Furthermore, transmission and unmarshalling at clients is only required for new objects or objects that have changed their value since the previous fetch. (At the implementation level, when a client issues a fetch it tells the server when it last received a copy of the space; the server then sends only what has changed since then.) This decision makes persistent spaces potentially more scalable than traditional IPC models, although actual performance will depend on their usage in real-world applications.

We expect that programmers will use this one-to-many semantics to their advantage, grouping related objects in a number of spaces published by each server; clients will then subscribe and fetch only the spaces in which they are interested. If a different semantics is required, then another mechanism should be used. For example, the Distributed Library Explorer described in section 7.1.2 uses a persistent space to propagate a relatively small

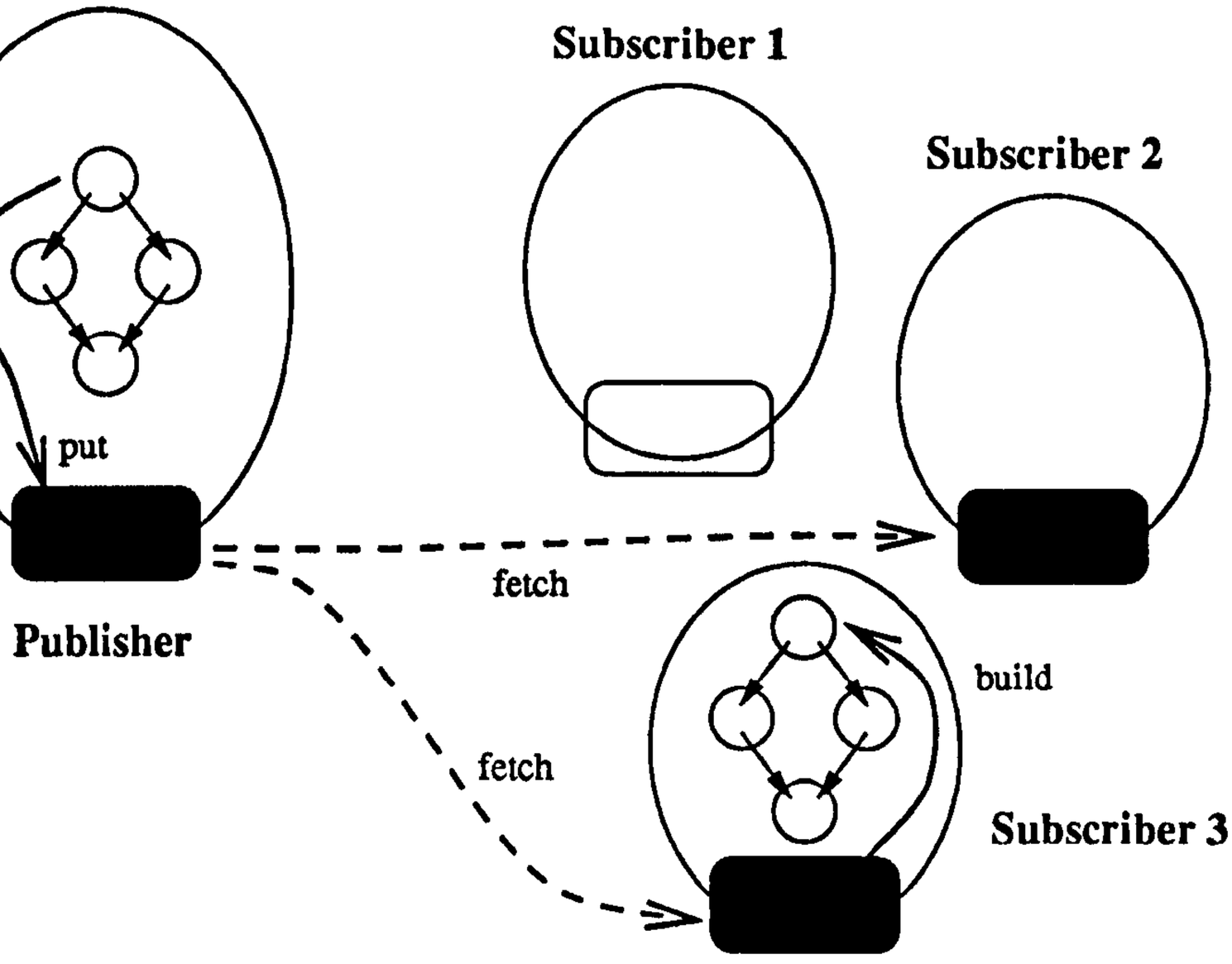

Figure 6.1: Overview of persistent spaces

data structure (500 KB) to all clients, but also makes use of RPC to access a much larger database (10 MB) that would be duplicated on all clients (see section 6.3.2).

The objects re-built locally in the client are *read-only copies*, otherwise a full replication protocol would be needed to maintain all copies of the space consistent. (The current implementation of persistent spaces does not enforce that local copies are immutable. We recognize this situation can lead to problems if the application programmer cannot logically separate objects belonging to a persistent space from other normal objects in the store. However, there seems to be no major difficulty with incorporating this functionality in the future, which will also not change the basic semantics of the model.)

In addition, objects put into the persistent space by the server will *not* be propagated to the clients automatically by the mechanism. The *fetch* operation should be called explicitly by each client to refresh its own copy of the space. Again, another IPC mechanism should be used if clients are required to update remote objects or access always the up-to-date version of the object. (This could be confusing, but the experience with the Library Explorer suggests that application programmers clearly separate each IPC mechanism and use them according to their semantics, see section 7.1.)

A server can put (top-level) objects into a space at any time. There is only one explicit put operation for each top-level object put into a space, but the entire graph of objects reachable from that top-level object will be put into the space as well. Fetch does not imply any put; it will simply bring copies of the objects currently in a space to the client that made the request.

It is likely that for many applications a small number of spaces with a few top-level objects — but potentially many nested objects — will be published by each server. (The right number of spaces and objects obviously depends on the application being built, see section 6.2.5).

Finally, persistent spaces were *not* designed to prevent an application programmer from putting objects with extremely large transitive closures into a space, like the persistent root. We hope application programmers will use their own experience and other techniques, such as substitution, to avoid putting large parts of the store into a persistent space. (Although we do propose an operation to test the size of the transitive closure before deciding to put an object into the space, see end of section 6.4.3.) On the other hand, persistent spaces are adequate for relatively large transitive closures because they maintain sharing semantics and transmit objects incrementally.

6.2.2 Failure Handling

A failure code is returned immediately to the client application program if the mechanism cannot perform *fetch* in some "reasonable time" defined by the application programmer. No other operation on a persistent space involves the network or another store, with two main advantages.

- *Failures are returned to the application program when fetch is called*, not arbitrarily when access to the object is eventually required by the application.
- *The client program can respond to failures based on knowledge of the application*, not in some "average" way that usually means no more than retrying the operation for a few times and aborting if it continues to fail. For example, the application can maintain a list of alternative or *mirror sites* to point if the main site does not answer.

In summary, a single and explicit remote operation means that partial failures and network load correlate with the constructs in the application. This means programmers know when the application will pay the price for a remote access and where it can fail so that programs are written to catch them and react accordingly.

6.2.3 Server API

The interface to persistent spaces at the *server* is based on the following operations (see figure 6.2). These operations are all local, that is, they are performed by the server alone without the involvement of any other store. (Operations *publish* and *put* will be better described in section 6.4.)

- *Publish* the space and give it a name. This operation creates the data structures needed by the persistent space.

- *Put* a pair <name,object> into the persistent space, even when the space is in use. This operation marshals the entire transitive closure of the object and stores the result (a byte array) inside the space. The object name will be used by clients to have access to this object in their stores (assuming that objects, like spaces, will have "well-known" names). Put occurs *incrementally* by adding only new objects, or objects that have been updated since the last put, into the persistent space — including all objects in the transitive closure of the top-level object being put into the space. If an object with the same name was previously put into the space, then the new value replaces the old value. *Put* also maintains sub-structures, preserving sharing semantics between *put* operations. For example, if objects A and B refer to C, then if A and B are both *put* into a space, C is only *put* (marshalled) once. (Sharing semantics are *not* preserved across persistent spaces, see section 6.3.3.)

- *Drop* an object from a persistent space by name. The object becomes unreachable from the persistent space and will not be available next time the space is fetched. (Since only top-level objects have names, drop does not apply to nested objects; these will remain in the space if they are reachable from other top-level objects.)

- *Unpublish* the space so that a subsequent *fetch* from a client will fail. The contents of the space — marshalled objects and auxiliary data structures — can be garbage collected.

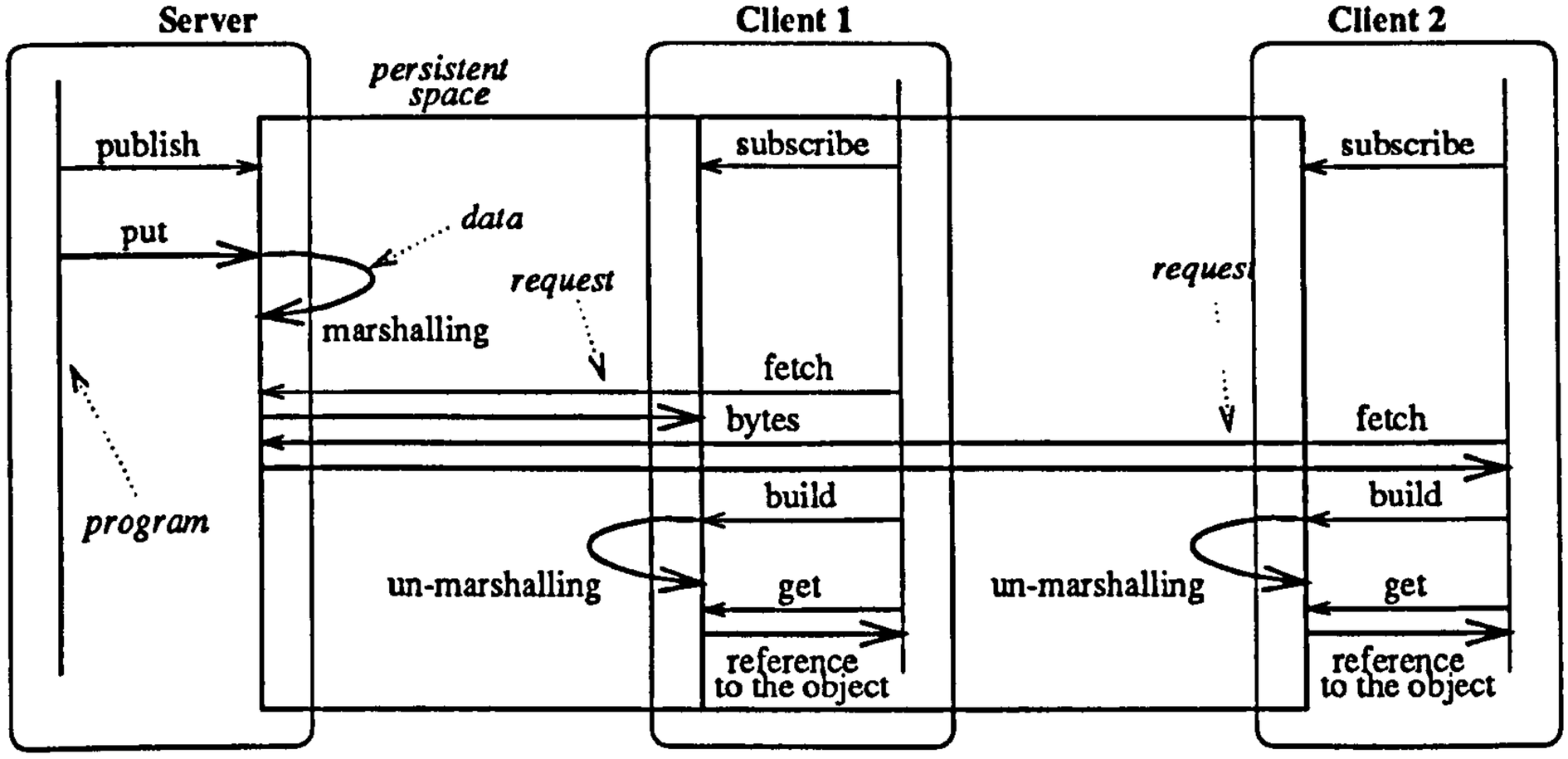

Figure 6.2: Overview of the operations in a persistent space

6.2.4 Client API

All operations at the *client*, except *fetch*, are also local. (Operations *subscribe*, *fetch*, *build* and *get* will be better described in section 6.4.)

- *Subscribe* to the persistent space by giving its name and the address of the server that published this space. (This information is either "well-known" already or can

be obtained on demand from a binding server using Napier/RPC, see section 4.2.3.) The operation does not check if the space has in fact been published to avoid a second remote operation; an impatient client can always fetch the space right after subscription to check if the space has been published.

- *Fetch* the contents of the space. After the first fetch the operation is performed *incrementally* since only the differences (deltas) to the previous fetch from that client are actually transferred. If it has not succeeded after a period of time defined by the client program, *fetch* returns an error code and a textual message (e.g., "server **alderney** not responding").

- *Build* constructs local (but read-only, see section 6.2.1) copies of all objects put into the space by the publisher. It does not return any reference to the objects built to separate unmarshalling from actual use (see below). Based on the incremental *put* and *fetch*, this operation also works incrementally because it unmarshals only the differences transferred from the server. Common sub-structures are maintained; in the example for the *put* operation above, A and B will still share C in all clients. If an object was built before, then the new value replaces the old value.

- *Exists* accepts an object name and returns `true` if an object with that name has been put at the server, fetched and rebuilt by the client. (The server address is not needed because the object is a local copy of the original (remote) object put into the space.) *Exists* can be used before *get* (see next operation) when the client is not sure if an object belongs to a space.

- *Get* accepts an object name and returns a local reference to a read-only copy of the original (remote) object put into the space. This is a very fast operation because the value has been unmarshalled already during *build* and there is an index — maintained by the persistent space — from object names to object values. (Only top-level objects have a name.)

- *Un-subscribe* discards the persistent space at the client and all its contents. (The server uses *un-publish* to discard its own, master copy of the persistent space.) Subsequent operations on this space become invalid. Objects in the space referred by other objects in the client program are not removed because of referential integrity, but objects not reachable from the persistent root are candidates for garbage collection.

In addition to this basic set of operations, others can be later added to the model, for example, to get the names of all spaces published by a server, to check if a space with some name is currently published, or to check if an object has been put or updated since the last fetch. However, these and other extensions will be added only if application programmers show a real need for them and they maintain our original requirement for simplicity.

6.2.5 Summary

Persistent spaces are designed to work well with:

- many inter-dependent objects per space because shared structures are preserved within spaces;
- large number of clients because an important part of the cost is amortized by the number of clients; and
- geographically distributed applications linked by poor network connections because only the differences are transmitted at well-defined times.

On the other hand, the limitations of persistent spaces should also be taken into account:

- all clients receive (incrementally) a copy of the entire space;
- only a relatively few spaces per server can be used because shared structures are not preserved across spaces; and
- objects rebuilt in the client cannot be updated.

A formal analysis of the CPU, bandwidth and space costs of persistent spaces is not included in this thesis. Instead, we have built a (real) distributed persistent application using persistent spaces and measured both the mechanism and the application (see sections 7.1.2 and 7.2.3).

6.3 Interactions with Other Mechanisms

Even though persistent spaces are a *complete* IPC model, they are not to be seen as the only solution to the problem of sharing objects in a distributed persistent environment. Instead, persistent spaces are just another library available to programmers and should be used in combination with other mechanisms—local and distributed—depending on the desired behaviour for the application, system characteristics, failure rate, and various costs.

6.3.1 Local Mechanisms

Persistent spaces are built on top of the language and as a result both the compiler and run-time system for the language remain unchanged. This means that existing features in the language, such as *garbage collection*, will not interfere with persistent spaces more than with any other local program (and vice-versa). For example, when a persistent space is un-subscribed from by a client, all objects in that space and other support data structures become candidates for garbage collection (except those objects that are now reachable from the persistent root).

Other programs running concurrently with persistent spaces can interfere. For example, since the *put* operation is not atomic (see transactions below) there is a potential for changing sub-objects while these are being marshalled. The integration between concurrency and distribution is already difficult, and its problems are further amplified when persistence and partial failures are taken into account (see, for example, the work by Munro [Mun93]). Application programmers should themselves take care to avoid this situation by using the normal mechanisms for concurrency control available in the language.

Although at present transactions are not included in the model, both local and distributed transactions can be useful in the context of persistent spaces. Some integration can be achieved already by the application programmer. For example, the use of a local transaction at the server can support a number of *put* operations as an atomic action.

There are mainly two reasons for not incorporating transactions into the model: it is not clear whether the advantages provided by transactions always compensate the price for using them (e.g., a decrease in performance and a more complicated interface); and many persistent applications run (or can be changed to run) within a transaction already. Simplicity was also the main reason to omit transactions from Digital's Network Objects [BNOW93].

Distributed transactions are also of interest to persistent spaces, especially as a support mechanism to transmit a space atomically between the the server and its clients. For example, a message passing mechanism — in which simple byte arrays are sent efficiently and reliably between two programs, see section 2.3.8 — could be used as the transport protocol.

6.3.2 Migration by Substitution

Substitution is orthogonal to persistent spaces; they complement each other and can (even should) be used together. Substitution is useful when an object needs to be put into a persistent space but it includes in its transitive closure one or more objects that cannot migrate — for example, an object that only make sense locally, such as a file descriptor (see section 5.2.4 for more examples).

As an example of the interaction between substitution and persistent spaces, the Distributed Library Explorer presented in section 7.1 makes use of the *three mechanisms* proposed in this thesis. The application is separated into a client program and a server program. The client is installed by some conventional mechanism and is responsible mainly for the user-interface. The use of all three mechanisms is exemplified below.

1. The server maintains a large database, but ships to the clients an index in a *persistent space* to speed-up most accesses to that database. The index is a medium-size data structure, much smaller than the database itself.

2. Two procedures that are used by the index *migrate by substitution* since these procedures are installed together with the client and will never change. Substitution also

avoids transmitting the procedures, a relatively expensive operation when compared with other (simpler) data types.

3. Access to the database itself—a much less frequent operation than using the index—uses a *remote procedure call* so that clients do not need to copy and maintain the (large) database.

The interested reader is referred to section 7.1 where a more detailed explanation is presented, including diagrams and measurements.

6.3.3 Multiple Spaces per Store

This section discusses the interaction between two or more spaces published by the same server. The most important point is that—in contrast to a single persistent space that maintains shared sub-structures between successive migrations—two persistent spaces do not maintain these relationships between them.

As an example, imagine two objects A and B, with A referring to B. The programmer would like to put A and B into separate spaces, respectively called *green* and *blue*, with an intention to publish them independently. If object A is put into the persistent space *green* first, then the entire transitive closure of A (including B) is marshalled into that space.

This is not the desired behaviour, so the programmer tries instead to put B first into the persistent space *blue*, then A in *green*. The hope is that A will use the copy of B in *blue*. However, if A could be marshalled into a persistent space without part of its closure, then *build* would depend on the relationship between persistent spaces and the order in which objects were put into those spaces. For example, an error would occur when building *green* if *blue* had not been fetched and built before. Dependencies like this can be acceptable for two spaces and one client, but do not scale for many inter-related objects and a large number of stores and spaces.

What really happens when B is put first into *blue*, then A into *green*, is that *B is duplicated in both spaces*. It was decided to prevent sharing between persistent spaces in order to keep the model simple to understand and use. Otherwise, complicated dependencies between persistent spaces could easily be created that would not be understood by average client application programmers—especially because these dependencies would be created by another programmer in the (server) store. On the other hand, this separation between persistent spaces also gives clients freedom to subscribe to any sub-set of all persistent spaces published by a server, depending on their particular requirements.

If only one version of B is to be published by the server, then *only one space should be used*. Alternatively, two spaces can be used but care should be taken so that B is put into both spaces simultaneously (ideally within a server transaction) and clients also fetch and build these spaces simultaneously (ideally within a client transaction).

Considering the highly inter-connected nature of persistent objects, many objects in the

store can potentially be duplicated in more than one persistent space. However, the Library Explorer example application presented in section 7.1 suggests that application programmers are able to *isolate* objects intended to be published. Substitution can help as well by cutting links to other objects that already exist in all stores. Finally, better visualisation tools for understanding the relationships between persistent objects [Lav95a, Lav95b] can also help to organize which objects should go into which persistent spaces.

Higher-level operations could have been incorporated into the model to give some guarantees between spaces, for example, to return a warning if an object being put into a space is already published in another space. However, it was decided to maintain the simplicity of the model because it is not clear whether this extended semantics is required by many persistent applications.

6.3.4 Clients as Servers

Persistent spaces only connect one server to many clients. On top of this, application programmers can themselves provide another extension to the model: one persistent space is fetched by a client store and then (some of) its objects put again into another persistent space. This store is acting as a client in one case and as a server in the second case.

The objects put into the second space are *copies* of the original objects put into the first space. Thus a *fetch* operation applied to the first space will *not* change the values of the objects in the second space. This permits stores to behave as intermediaries of data, filtering or doing any other processing to collections of objects (bulk types). Explicit copying between spaces is appropriate for mostly disconnected stores, the intended target of persistent spaces. If the application needs more tightly-coupled stores, then probably another distribution mechanism should be used.

6.4 Application Programming Interface

This section describes in detail the fundamental operations available for persistent spaces — *publish*, *subscribe*, *put*, *fetch*, *build* and *get* — which present a number of common design principles.

- In order to emphasize the semantic differences between persistent spaces and other IPC models, namely RPC, the model presents *a new interface to the application programmer.* It would be dangerous if programmers confused persistent spaces and RPC since they have different semantics. This separation is especially important in applications that take advantage of preserving sub-structures and incremental migration as provided by persistent spaces.
- Another objective of this API is *to help programmers build distributed applications that react to partial failures* by using their specific knowledge about the application

semantics. *Fetch*, the only remote operation, is explicitly initiated by the programmer to provide a *single point of failure.*

- Finally, persistent spaces were designed *to present a small set of well-defined, very simple operations* that can be understood and used by normal application programmers. The intention is to support the development of distributed persistent applications without extensive knowledge of distribution.

The example used to illustrate the API is the Distributed Library Explorer taken from section 7.1. The Explorer is a distributed persistent application that maintains a *database* of information about software libraries and an *index* to access that database. The database and the primary copy of the index reside in a server at `/local/fide/users/mms/server` on a machine called `alderney`. In order to reduce response time on typical queries, the index is published by the server in a persistent space called "Explorer" that can be subscribed by any number of client stores.

6.4.1 The *Publish* Operation

First, the server needs to create a new persistent space before putting any objects into it (see example 6.1). The variable `pubexplorer` will be used later by the server to refer to this space.

```
! creates a new persistent space at the server
let pubexplorer = publishSpace( "Explorer" )
```

Example 6.1: Publishing a persistent space

Persistent spaces have application-level names, such as "Explorer", for simplicity. If the subscriber is being developed by other application programmers, then some other mean (such as e-mail) should be used to communicate the name of each persistent space. (Alternatively, a name server at a well-known location could be used, similar to that included as part of Napier/RPC 1.0 [MdS95a].) In order to use a persistent space, clients only need to know its name and the server address. (Authorization issues and security in general are not discussed in this thesis.)

6.4.2 The *Subscribe* Operation

A client needs to *subscribe* to a persistent space before using any objects *put* into that space (see example 6.2). The `publisher` variable identifies a remote persistent store anywhere in the local network — or on the Internet, if the domain is added to the machine name — and is created by applying the type constructor `RemoteStore` that groups a *machine name* and a *store path.*

```
! does nothing (type constructor used for convenience)
let publisher = RemoteStore( "alderney",    ! machine name
        "/local/fide/users/mms/server" )    ! store path

! creates a handle for the space
let subexplorer = subscribeSpace( publisher, "Explorer" )
```

Example 6.2: Subscribing to a persistent space

Subscribe does not check if the publisher store exists or if a persistent space called "Explorer" was published by that store. This check would require a remote call to the server and it is not needed since the first *fetch* will return this information anyway. The operation is necessary to define the variable that the client will use in all subsequent operations to refer to this space. (Internally, the data structures that will maintain the local copy of the space are also created.)

If the client program requires immediate confirmation of subscription, then the client can always attempt to fetch the space—a relatively inexpensive operation since the marshalling has already been performed—right after subscribing to it. (However, transferring the bytes may be still expensive depending on the space size. We suspect most applications will subscribe to the space just before the first fetch, so they will have a confirmation.)

6.4.3 The *Put* Operation

After *publish*, the initial set-up operation at the server, a number of objects can then be *put* into the space (see example 6.3). The variable `pubexplorer` was created previously when calling `publishSpace` in example 6.1.

```
! puts an object into the space
putObject( pubexplorer, "exploreridx", any(index) )
```

Example 6.3: Putting an object into a space

Put inserts (a reference to) the object in a map indexed by its name, marshals the value of the object and its transitive closure to a byte array, and stores the byte array in a data structure inside the persistent space ready to be sent to subscribers. Thus objects put into persistent spaces are just normal language objects because only copies of their transitive closures are actually stored in linear form within the persistent space. Marshalling at put time avoids repeating the marshalling every time a subscriber fetches the space, amortizing this cost over all potential clients.

Shared data structures are preserved between successive put operations—even between program executions by virtue of orthogonal persistence. For example, if two objects A and B are put into a persistent space, and both refer to a third object C, then *C will*

be marshalled only once. Application programmers can thus arrange to preserve sharing by placing all of a graph of objects in the same persistent space. The *fetch* and *rebuild* operations then honour this sharing.

Put also contributes to incremental migration. For example, if an object with the same name already exists in the persistent space, then the existing value is replaced by the new value. But if an object is *put* twice in the same space with exactly the same value, then nothing is stored in the space. (At the implementation level, some indication does need to be stored to keep track of put operations themselves.) In general the object has changed partially and *only the differences to the previous value are stored in the persistent space.* If many small objects are expected in the same persistent space, then—for semantic as well as for engineering reasons—a single bulk type can be used instead by the application programmer.

Marshalling is an expensive (thus long) operation, especially if the object is large or complicated—both likely in a persistent environment with a rich type system. *Put* attempts to reduce this difficulty with three characteristics that make it different from traditional marshalling.

1. *Put executes locally in the publisher store* without a live connection to any subscriber. This decision makes marshalling time limited only by local performance and reduces the dependency of the server on external factors (such as network load).

   The price to pay for this advantage is that persistent spaces are useful for only a certain class of distributed persistent applications in which many clients need to share the same set of objects made available by the same server.

2. *The marshalling cost is amortised over all subscribers.* The persistent space is prepared incrementally for each *put* operation and is simply copied when a client issues a *fetch*. This separation between marshalling and transmission also permits marshalling large objects when the system is lightly loaded, e.g. during the night.

   The price to pay is that *all clients will receive the same set of objects.* This is acceptable for the class of applications described in section 6.1.1, although in general other mechanisms will be needed to complement persistent spaces.

3. *Objects put into the space are guaranteed to be rebuilt in all clients* since *put* marshals the entire transitive closure of the objects.

   The price to pay is that potentially many objects not needed by any client will be marshalled, transmitted and rebuilt in all clients. (The problem can partly be countered by using more spaces, although this solution can only be used with objects relatively separated from other objects; on the other hand, the Library Explorer example in section 7.1 suggests this separation between objects may occur in some applications.)

The implementation of *put* (see section 6.5.1) is based on existing marshalling algorithms used in traditional RPC mechanisms. It may be argued that an application programmer

could build a mechanism equivalent to persistent spaces by constructing a list of objects to be transferred, marshalling the list, holding it on the server in its serialized form and sending it to any client that requests the list. The clients then unmarshal and use this list.

In fact, persistent spaces include the functionality just described above. However, we believe persistent spaces are a better approach for the class of applications described in section 6.1.1 because:

1. common sub-structures are maintained between migrations (though not between spaces) which is not supported by traditional pickling and serialization mechanisms [HL82, BJW87, BNOW93, Cra93, WRW96]; and

2. they support incremental migration by transmitting only the differences to the previous values put into the space (see section 6.5 for the implementation).

These features are crucial to support sharing of large, complex persistent objects between autonomous stores.

Limitations and Future Work

There is a potential problem with *put* and persistent spaces in general. Given the long-term requirement of persistence, the size of the data structures that implement a persistent space will monotonically increase with time. (In a non-persistent environment these data structures would be reset every time the program starts, but then application programmers could not take advantage of persistence.)

The operation *drop*—to withdraw an object from a space—is a partial solution to this problem. However, *drop* by itself cannot reduce significantly the amount of data in a persistent space since the byte arrays containing the marshalled objects cannot be deleted (incremental migration requires the previous versions of a persistent space in order to send only the differences). An effective way to remove excessive data accumulated over time from a persistent space is still a research issue.

On the other hand, more functionality not currently part of the model can be easily added later as part of future work.

1. *The put operation could return the space requirements (in bytes) for the marshalled object.* This number would give a useful indication of the costs needed to fetch the space. (Although this number is calculated already in the implementation, it is not returned to the application program.)

2. *The put operation could be made tentative*—in the sense that subscribers could not fetch these pending objects immediately—and be complemented by an *unput* operation to abort the last put. Then, a new operation called *commit* would execute

all pending puts within a transaction. These operations would give application programmers some "all or nothing" support without the cost and complexity associated with complete ACID transactions (see sections 2.3.8 and 3.3.13.

Finally, there are many reasons why *put* can fail. We have already listed some of these reasons (see sections 5.2.4 and 5.5.2) and how substitution can help in certain cases (see section 6.3.2). However, the general problem remains: how to detect and explain at runtime to the application program why an object could not be put into a persistent space.

6.4.4 The *Fetch* Operation

There is a need to *fetch* a persistent space only when a subscriber wants to access the latest versions of the objects put into that space (see example 6.4). The variable `subexplorer` was returned previously by `subscribeSpace` and `timeout` is the number of seconds after which the operation should be aborted if the space has not been completely fetched. If it fails, *fetch* returns an error code explaining why it failed (see below).

```
! copies the space to the local store
let result = fetchSpace( subexplorer, timeout )
```

Example 6.4: Fetching a persistent space from a publisher

Fetch transfers a set of byte arrays representing the objects put into the space since the last *fetch* was performed by this particular client. The operation works incrementally because only the additions since the last *fetch* are transferred, not its entire contents. For example, if no objects have been put into the space since the last fetch, then only a small indication explaining the space remains the same is sent to the client. In addition, for objects already in the space, only the differences between their old and new values are transferred.

The connection time between client and server is minimized because marshalling and unmarshalling — the expensive parts of migration — are either performed before (as part of the *put* operation) or after (as part of the *build* operation) transmission. It could be further reduced by compression. For example, the time required to transmit the *index* in the Distributed Library Explorer (see section 7.1.2) is only a small fraction of the time required to marshal and unmarshal that data structure. This is an important advantage of persistent spaces compared with more traditional RPC, in which all these operations are performed for each object transmitted.

Failure Handling

Fetch is the only truly remote operation and it returns a value that indicates if the transmission has succeeded or failed. The variable `result` in example 6.4 is a data struc-

ture—containing an *error code* and a *textual message*—based on our earlier research work [MdS95b]. The following are examples of error codes:

- `ConnectIgnored`—if the server does not respond;
- `NotListening`—if the server is executing but not listening to clients;
- `Timeout`—if transmission has not finished successfully after a certain period of time defined by the client program; and
- `MessageCorrupted`—if the network message received is invalid.

In addition, there are error codes specific to persistent spaces, for example:

- `SpaceNotPublished`—if the space has not been published by the server yet or has been unpublished already.

The textual message describes these codes in plain English and adds relevant information if appropriate, e.g., the name of the persistent space, the server network address and store path, number of seconds for the timeout, and so on. For example, a typical message is "Server `/local/fide/users/mms/server` on `alderney` not responding". The client program then uses the error code to take some action (if it has been programmed for that) or shows the end-user the textual message.

Future Work

The persistent space is transmitted as a set of byte arrays (excluding those the client has received before). There is one of these byte arrays for each *put* operation (see section 6.5.1 and 6.5.2). Although not currently implemented, these byte arrays could be written to a file and sent to subscribers by other means instead of direct program-to-program connection. For example, e-mail or ftp could be used, and even a physical medium such as floppy disks or tapes.

This alternative to *fetch* could be useful for disconnected or mobile users, who generally have limited bandwidth, expensive network links and/or unreliable connections. It could also be useful for security, performance, historical, management, or any other reason when a live connection is not appropriate.

6.4.5 The *Build* Operation

After fetching a persistent space, a subscriber unmarshals its contents using the *build* operation (see example 6.5). Copies of all objects put in the space before it was fetched by this client are built locally (except those that have been built before).

```
! unmarshals the space in the local store
buildSpace( subexplorer )
```

Example 6.5: Building a copy of the remote object locally

Build unmarshals all objects put into the space. The more flexible alternative of unmarshalling only a sub-set of the objects was excluded to maintain the simplicity of the model, and in particular to guarantee that a space can always be built after a successful fetch (see section 6.2). This is the reason why `buildSpace` does not need to return any value to confirm it succeeded. (In the current implementation, it was decided that severe errors just abort the program execution—for example, if the byte array is corrupted. This decision is typical of research prototypes and can be found in the Napier88 implementation itself.)

If the objects to be built refer to any objects unmarshalled in previous *build* operations, then *shared sub-structures are maintained.* The opposite is also true: if a shared sub-structure is re-built with a different value, then all objects that refer to this sub-structure will also share the new value.

The preservation of sharing together with incremental migration is the only realistic option if persistent spaces are to scale up to large numbers of complex, persistent objects. The measurements presented in section 7.2 show how the semantics chosen for persistent spaces significantly reduce the amount of data being transferred between stores when compared with transferring all objects put into the space for each *fetch* operation. (Unfortunately, the time for marshalling and unmarshalling remains a problem because in the current implementation the algorithms still have to *traverse* the entire transitive closures, even if no marshalling is actually performed.)

6.4.6 The *Get* Operation

The *build* operation creates local copies at the client of the objects put into the space at the server. However, the objects are hidden inside the space until the client program uses *get* to select a particular object from all those resident in the local version of the space. *Get* uses a data structure maintained internally by the persistent space to return a *local* reference to the object being requested (see example 6.6).

```
! makes an object visible to the target program
let index = getObject( subexplorer, "exploreridx" )
```

Example 6.6: Getting a reference to an object in a space

After the operation, the variable `index` now holds a reference to the object that implements the Explorer index. The current execution is simply aborted if no object with that name was put into the space, fetched and built. The client program can always use another

operation called *exists* (see section 6.2) to confirm if an object with that name is currently available locally in that space.

The value returned by `getObject` has type *any*, the infinite union of all types (see section 3.2.2). The client program then has to project the variable into a usable type. This *semantic indirection* between the persistent space and the application program — selection by name plus type projection — is useful to make the programmer aware that `index` is only a read-only, local representative of the actual *index* (primary copy) in the server.

6.4.7 Conclusion

In this section, the main operations available for persistent spaces were described and their semantics discussed. When appropriate, limitations of the existing model and suggestions for future work were also given.

The next section presents an implementation for persistent spaces in a persistent programming language. A practical example application will be described in chapter 7, together with performance measurements for both persistent spaces and the example application. Future work is presented in the context of the entire thesis in chapter 8.

6.5 Implementation

A prototype of persistent spaces was built to validate the proposed IPC model and the semantics chosen for the operations available to the application programmer. We are especially interested in the implementation for a number of specific reasons: to confirm the *feasibility* of the model; to provide a platform for *performing measurements*; to *build and test example applications*; and to *develop the initial design* through our own experience and feedback from other application programmers.

The prototype was built in Napier88, a persistent programming language described in section 3.2. It modifies and extends an existing mechanism to copy objects of any type between two Napier88 stores [Mun93, KBC+94]. It should be clear that Napier88 is used only as an example; persistent spaces can be implemented in *any* equivalent programming language. In particular, the implementation language should support the following features: orthogonal persistence, dynamic binding, and support for communication between autonomous stores.

Persistence is a requirement because the data structures used in the implementation need to survive program executions. It could be possible to implement persistent spaces in a non-persistent programming language, but *persistence simplifies the implementation enormously* as these data structures persist automatically by reachability from the persistent root. On the other hand, a non-persistent implementation would *not* be very useful since persistent spaces were designed for a persistent environment.

This first implementation of persistent spaces *was not optimised* in terms of reducing costs and increasing efficiency. The reason is that persistent spaces will evolve with time and the implementation will change to reflect the new semantics. It is not a good use of limited research time to optimise an implementation that will be short-lived and used only in a small number of example applications. Nevertheless, measurements presented in section 7.2 show that acceptable performance can be achieved if we take into consideration the relative speed of Napier88.

Overview

The section is divided into three parts that correspond to the three most important operations on persistent spaces.

1. *Put*—Marshals the value of an object into a byte array in the server store.
2. *Fetch*—Transmits a set of byte arrays (one for each top-level object put into the space) from the server to the client store.
3. *Build*—Unmarshals the byte arrays received from the server to construct local copies (in the client) of the objects put into the space.

These operations include the traditional steps that are needed to migrate an object by copy between programs executing in separate address spaces [HL82, BN84]. (Similar algorithms for marshalling and unmarshalling cyclic and shared data structures were independently developed at the same time to support persistence, cf. PS-algol [ABC+83].) The implementation of persistent spaces *extends these algorithms* and *uses orthogonal persistence* to achieve the properties described in sections 6.2 and 6.4.

Objects with type at the *language level* (Napier88) are implemented by a number of untyped, *run-time system* objects that contain the object's value plus control data such as constancy and type information [CBC+90a]. The implementation of persistent spaces uses a special version of Napier88 (the npb compiler) that gives access to these lower-level, run-time system objects. (Language-level objects cannot be used directly because persistent spaces need to have access to lower-level information about the objects being published and also because it is much more difficult to work at the system level above the type-safety net provided by the normal Napier88 compiler.)

These objects in general form a cyclic and shared data structure. Marshalling is the process of writing this data structure to an array of bytes so it can be copied to another store. The *put* operation accepts a language-level object and marshals its run-time system objects. At the client, the *build* operation unmarshals the byte array to construct the graph and then rebuilds a typed, language-level object again.

6.5.1 Marshalling

The marshalling algorithm traverses the graph of objects in a recursive depth-first order, dumping all objects it visits (see below). The algorithm starts by calling *dump* to the top-level object and iterates through three phases for all objects in its transitive closure:

1. *dump*—if the object has not been visited yet, marks this object as *dumped*, outputs the object (see below) and dumps all objects it refers to (its nested objects);
2. *output*—exports information about this object's nested objects (graph structure) and exports its data; and
3. *export*—the bytes received as an argument are appended to an *export buffer*, which is the byte array that will be copied to client stores.

Data Structures

There are four main data structures needed to implement each persistent space: `putNumber`, `objectNumber`, `remembered` and `mapOfExportBuffers`. These are described below.

`putNumber` is an integer incremented every time the operation *put* is called for this persistent space.

`objectNumber` is an integer incremented every time an object is visited. It is used to identify objects in their linearized form. The first object marshalled is identified by the number 1, the second by 2, and so on, across *put* operations and program executions (that is, `objectNumber` is persistent and is never reset). When a client fetches a space, it sends the number of the last object received by that client and for that space. The server then sends back to the client only *new objects put into the space* and *updates to existing objects* that this particular client has not received yet.

`remembered` is a bulk type that stores the following properties for each object visited by the marshalling algorithm.

- `objnb`—An integer that identifies the object in the export buffer. This is the value of `objectNumber` at the time the marshalling algorithm first knows about its existence, either because it is about to be dumped or it is referred to by another object (see below for details).
- `pntrold`—A reference to the latest marshalled value of the object. When an object is visited and has been dumped in a previous *put* operation, its current (actual) value is compared with the value pointed to by `pntrold` to check if the object was updated in the meantime.
- `lastvisit`—An integer that was set to the value of `putNumber` when the last visit occurred. An object is defined as *visited* if the value of `putNumber` is equal to the value

of `lastvisit` and this means the algorithm has already passed by this (remembered) object during this *put* operation.

- `dumped` — A flag that tells if the object has been dumped already. `lastvisit` and `dumped` are not redundant because objects may have been dumped in a previous *put* operation but not visited yet by the current *put* operation, and they may have changed value in the meantime.

The `remembered` data structure is implemented as a list because in a persistent language it is not trivial to index a map by references to objects. The difficulty resides in the combination between persistence, type-safety, and garbage collection. A reference in this environment cannot be manipulated at the language level (like in C++) for safety reasons. (Persistent object identifiers are a possible solution, see section 8.2).

For a first implementation of persistent spaces we decided to search `remembered` linearly. The measurements presented in section 7.2 show that performance is acceptable, although it will not scale for a large number of objects. Another problem with the prototype is the space requirements of `remembered`, especially the need to store a copy of the object value when it was *put* into the space. This is the price to pay for incremental migration, although it can be reduced in future implementations by using version numbers, compression, or hashing schemes.

Finally, `mapOfExportBuffers` is a map indexed by object number to store *export buffers* (see figure 6.3). An *export buffer* contains a vector of bytes and its length, and represents the objects marshalled for one *put* operation. An *object number* is the number of the first object marshalled in each export buffer. (There is no index to access the object directly by its number because the marshalling algorithm never needs to get the object value given its number, only the unmarshalling algorithm.) When a client fetches the space, the implementation uses the last object number received by the client to transmit only the export buffers not received by that client yet.

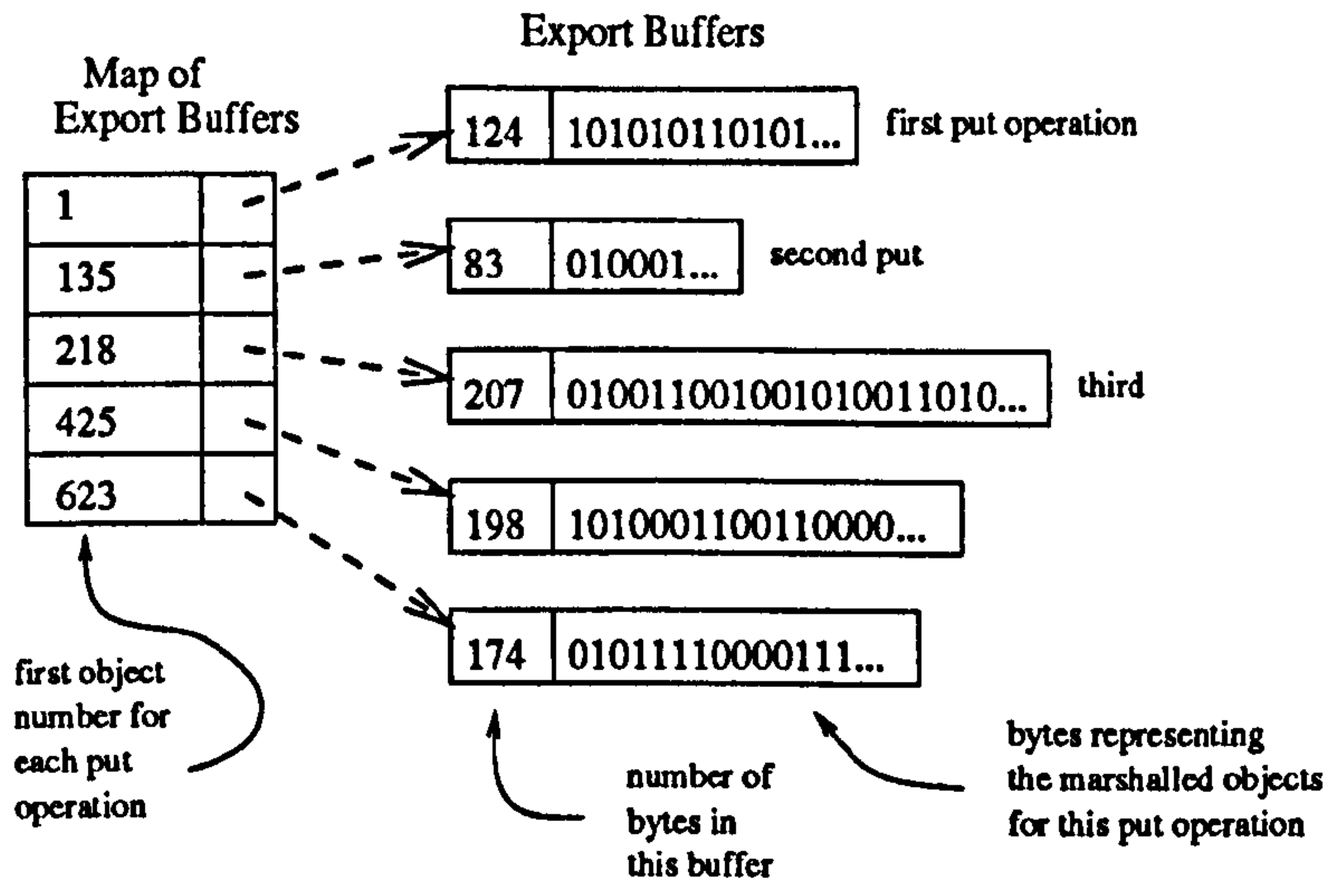

Figure 6.3: Map of export buffers

Main Procedures

The top-level procedure called by *put*, after getting the implementation object for the language-level object passed as an argument, is `dumpObject` (see example 6.7). The procedure distinguishes between: *new objects* that are output and its nested objects dumped; and *remembered objects* that have been dumped in a previous put operation. Furthermore, a remembered object can still have the same value or changed value since it was dumped.

If a remembered object has not been visited by this *put* operation, then all its nested objects are dumped as well since an object may have not been changed but its nested objects may have. (This means the full transitive closure of a top-level object is always traversed, although only objects in the transitive closure that are new or have changed value are actually written to the export buffer.) In addition, `dumpObject` outputs a remembered object again if it has changed value since the last visit (including objects that are remembered but have never been dumped, see below).

```
If the object is not in remembered, then ! never seen before
    insert the object in remembered
    set dumped to true
    call outputObjectHeader( objectNumber )
    add 1 to objectNumber
    call outputObject ! exports the object's value
    call dumpNestedObjects ! visits its sub-objects
else ! dumped already
    if it has not been visited already, then
        set lastvisit to putNumber
        if it has not been dumped already, then
            if the object has changed value, then
                set dumped to true
                duplicate the object value to pntrold
                call outputObjectHeader( this object's number )
                call outputObject
            call dumpNestedObjects
```

Example 6.7: Pseudo-code of `dumpObject`

`outputObjectHeader` appends to the export buffer the minimum information needed to build a skeleton of the object remotely (see example 6.8) while the sub-objects and data of the object are actually exported by `outputObject`. The procedure `exportBytes` just appends the data received as an argument to the current export buffer (implementation not described).

```
call exportBytes( object number received as a parameter )
call exportBytes( number of nested objects )
call exportBytes( size of the object )
```

Example 6.8: Pseudo-code of `outputObjectHeader`

`outputObject` exports the references to the nested objects as numbers (their identification in the export buffer) and its own data (see example 6.9). (`outputObject` cannot simply dump the nested objects as well because they may have to be dumped independently of their parent object, see `dumpObject`). Nested objects that have been dumped already (in this or in a previous *put* operation) have a number already; all others receive a new number and are remembered (but not as being visited).

```
For each reference to a nested object
    If the object is not in remembered, then
        insert the object in remembered
        add 1 to objectNumber
        call exportBytes( objectNumber )
    else
        call exportBytes( this object's number )
call exportBytes( this object's data )
```

Example 6.9: Pseudo-code of `outputObject`

`dumpNestedObjects` is called by `dumpObject` and just dumps all nested objects of a parent object (see example 6.10).

```
For each nested object
    call dumpObject
```

Example 6.10: Pseudo-code of `dumpNestedObjects`

6.5.2 Transmission

A persistent space is transmitted to clients as part of the *fetch* operation. *Fetch* is implemented as a remote procedure call (RPC) at the server that accepts as an argument the *last object number* received by that particular client and returns the set of export buffers not yet received by the client.

For example, imagine the persistent space depicted in figure 6.3 for which five put operations have been performed. Now consider that a client fetches this space and that the last number received by that client is 424. In this case, *fetch* transmits only the last two export buffers (indexed by 425 and 623) because the object number 424 received from the client says that the other three export buffers (indexed by 1, 135 and 218) have been received already by that client. It will transmit a different set of export buffers to other clients, depending on their last object number received.

6.5.3 Unmarshalling

Unmarshalling is implemented as part of the *build* operation. It is the opposite to marshalling: rebuilding (in the client) copies of the objects put into the space from the byte arrays sent by the server. Since marshalling and unmarshalling are so similar, only the three main differences between them will be described.

1. While for marshalling the `remembered` data structure contains the objects visited, a similar data structure now contains the objects already unmarshalled. But while for marshalling there is a linear search by object reference, for unmarshalling the data structure is a map indexed by object number. (In general, in a type-safe persistent language it is not possible to index objects by reference, see sections 7.2.3 and 8.2.2 for details.) This makes unmarshalling potentially more scalable than marshalling — although the measurements presented in section 7.2.3 do not show any significant difference in performance (perhaps due to the implementation of map).

2. During unmarshalling there is no need to keep the previous value of the object (the `pntrold` field in `remembered`) because each *build* operation simply replaces the updated objects with their new values. There is also no need to keep the `lastvisit` and the `dumped` fields for similar reasons.

3. Because marshalling proceeds recursively with export on first visit, unmarshalling has to create objects for which their sub-objects are not created yet. (Since these are cyclic data structures, it is impossible to start at the "bottom" of the graph.) This requires special privileges because it violates type-safety and referential integrity. However, when unmarshalling finishes all references have been established again, so that persistent spaces remain type-safe from the application programmer point of view.

The last item is the only one that deserves a better discussion. The code to achieve this — based on *pending lists* — is described below.

Pending Lists

As an example, imagine that an object was *put* into the space and that as a result the graph of objects depicted in figure 6.4 was marshalled. Using the algorithm presented in

section 6.5.1, the marshalling order for this graph is A, B, D, C. At the client, there is a difficulty because when object A is to be built there is no object B or C to refer to yet (the same happens with B and C referring to D).

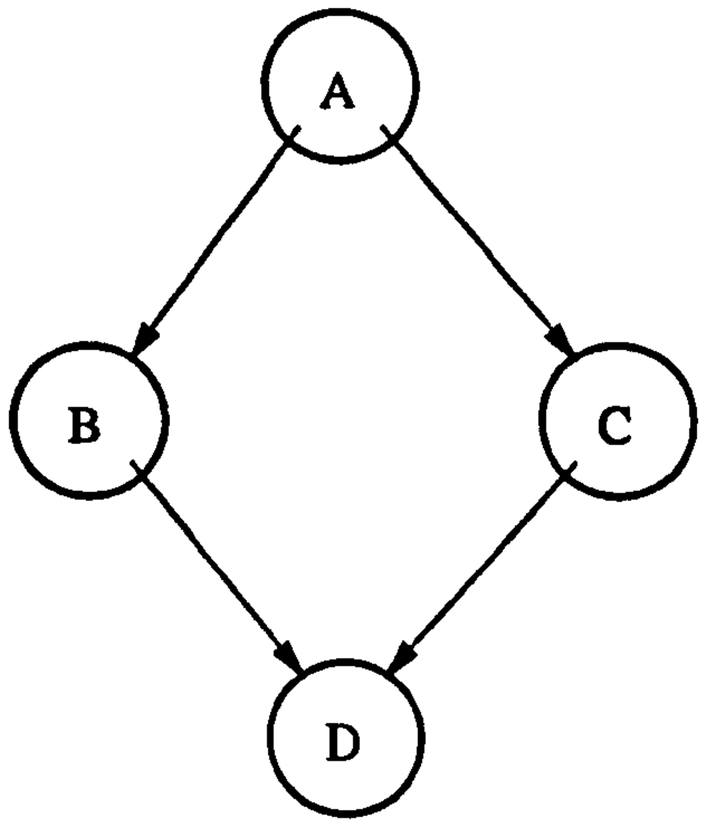

Figure 6.4: An example of a simple object graph

Even if the graph had been marshalled recursively with *export on return* (that is, D, B, C, A) the problem would remain because, in general, the object graph has cycles. For example, imagine B and C pointed to each other; objects would still have to be built pointing to objects that are not built yet.

One solution to this problem is to create a *pending list* for each object that is needed but is not yet built. When that object is built, its pending list is used to update all objects that were waiting for this one. All future references to this object can then be assigned directly because the object now exists.

Figure 6.5 illustrates the status of the pending lists after objects A, B and D have been built. Object A was created first so, at the time, two pending lists were created for B and C (the objects referred to by A). These lists contain offsets in object A where the reference to objects B and C will have to be written later when these objects are built. The pending lists of B and D have already been removed because objects B and D have been built already (dashed lines). The pending list for object C shows that object A is still waiting for this object to be built (full lines). The pending list for A is empty because no other object refers to A.

These pending lists may grow considerably in size, depending on the complexity of the object graph. However, when the last object is built—and all other objects that point to it have been updated—all pending lists should be empty. Referential integrity and type-safety is restored and these temporary pending lists can be garbage collected.

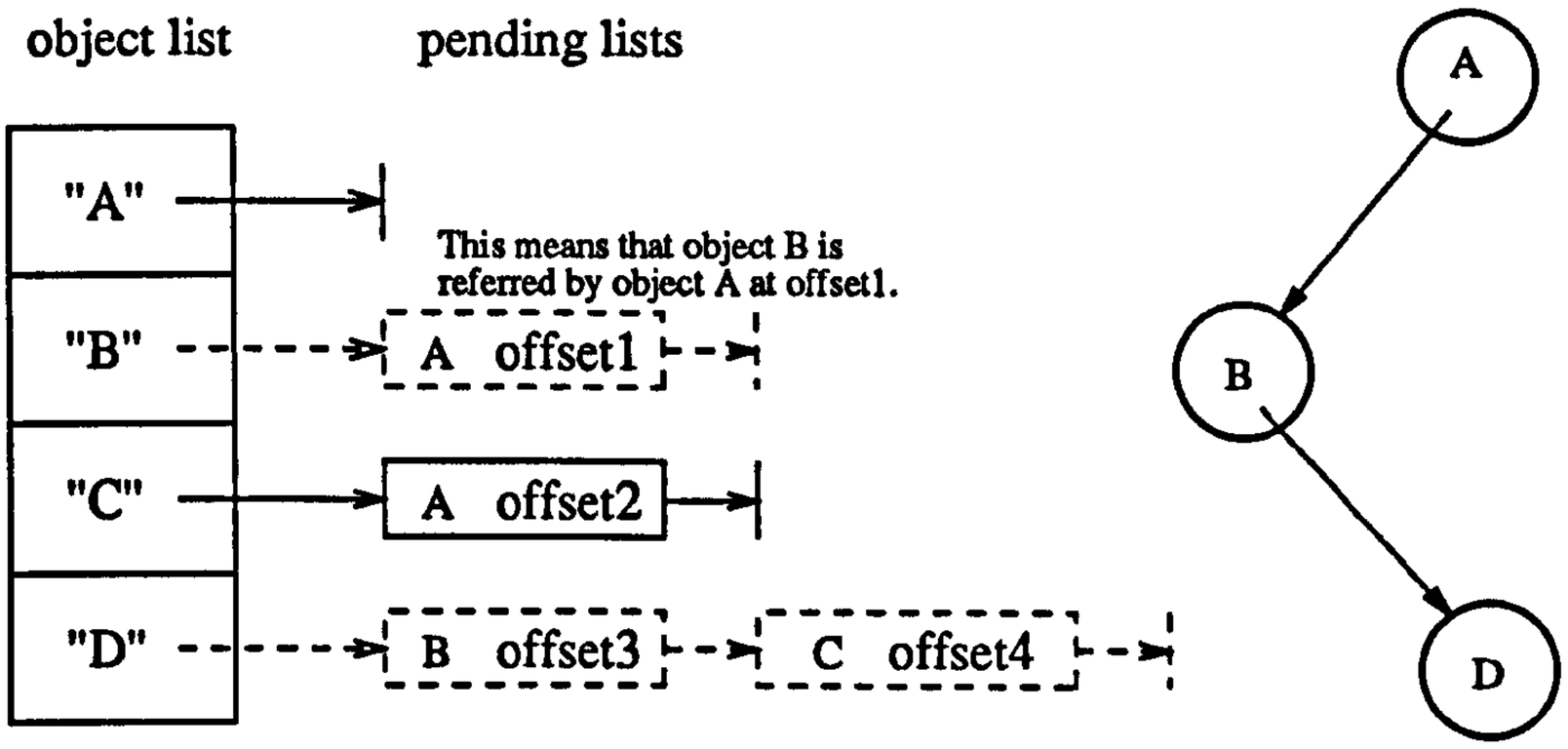

Figure 6.5: The *pending list* during an unmarshalling operation

6.6 Related Work

This section compares persistent spaces with related IPC models: *message passing*, for exchanging values explicitly between programs; *tuple spaces*, in which tuples can be put into, and retrived from, by any program in the distributed application; and *replication protocols* that maintain consistency between several copies of an object. (Systems that support "transparent distribution" like Emerald [BHJ+87] are not included in this survey because they are based on remote references instead of copying.)

Summary of Persistent Spaces

The main advantages, limitations and problems of persistent spaces are first summarized in order to better understand the similarities and differences between them and other IPC models.

The advantages of persistent spaces have been extensively described throughout the section. (All features that belong to *future work* are excluded from this list.)

- *Simple model based on publish and subscribe*—One server puts objects into a space and they are then available to any number of clients to fetch.
- *Marshalling cost is amortized over all subscribers*—Marshalling is performed only once when the object is *put* into the space. *Fetch* just transmits the byte arrays representing the marshalled objects.
- *Explicit refresh from clients*—Network load and partial failures only occur when *fetch* is called and cease after it finishes, so network activity is correlated with program constructs.
- *Minimum connection time*—Marshalling and unmarshalling are respectively performed before and after transmission by other (explicit and local) operations, while

only the differences from the previous state of the space are transferred (incremental migration).

- *Sub-structures are maintained*—Objects belonging to the same space do not duplicate shared objects within the space and local sharing semantics at the server are rebuilt in the clients.

- *Autonomy between client and server*—Autonomy is fundamental to reduce network load and failures caused by dependencies between stores and to permit local evolution, management, and so on. Furthermore, autonomy is especially important in a persistent environment because dependencies survive program executions and accumulate over time.

- *Composable with other mechanisms*—Persistent spaces can be used together with existing features of the persistent language (such as transactions) and other IPC models (such as RPC and substitution).

The main limitations of persistent spaces are now summarized.

- *One space for all clients*—Clients cannot choose which objects to fetch: they all get all the objects put into a persistent space. This design option is the basis for the simplicity of the model and most of its other advantages, but many clients will fetch objects they will never use.

- *Clients have access to read-only copies*—Another IPC mechanism has to be used if the application requires a client to update the original (remote) object in the server store. For example, a second persistent space in the opposite direction could be used. Again, this decision simplifies the model but only applies if the application (or at least part of it) belongs to those described in section 6.1.1.

- *Distribution becomes visible*—Some extra programming effort is required to *fetch* the space compared with "transparent distribution". However, this visibility can be considered both a limitation and an advantage, since programmers need to know about and use distribution if the application is to react to partial failures (caused by that same distribution).

In addition to these limitations in the model, there are a number of problems with the current implementation that need to be investigated (see section 8.2.2).

- *Space required in the server and client stores*—The prototype requires relatively large auxiliary data structures, in particular the copies of the objects maintained by the persistent space at the server (pointed by `pntrold`, see section 6.5.1) to check whether objects have changed since they were exported last time. (There is a large scope for improvement here, for example by using fingerprinting, see section 2.3.6.)

- *Poor "absolute" performance*—Performance can still be greatly improved in *absolute* terms, although it is *relatively* acceptable when compared with its working environment (see sections 7.2.3 and 8.2.2).

These limitations could have been significantly reduced by improving the algorithms and using a more careful choice of data structures. We opted instead to concentrate on the model itself and leave enough space in the implementation for future experiments. On the other hand, a radical improvement is always possible by re-implementing persistent spaces within the run-time system without compromising type-safety (see section 8.2).

6.6.1 Message Passing

Message passing is a very simple IPC model in which *values are exchanged explicitly between programs.* The basic model is asynchronous (see section 3.3.10) but message passing can also be synchronous: one program sends a message to another program that is listening to the network waiting for incoming messages.

Persistent spaces are very similar to asynchronous message passing because the aim is also to migrate groups of objects explicitly between autonomous programs. However, persistent spaces are designed for complex, persistent objects: they help the application programmer *organize the transmission*; *provide incremental migration*; and *maintain sharing semantics* between migrations (even between program executions).

Message passing products such as IBM's MQSeries [IBM94, IBM95] only support simple data structures (like records of primitive data types) or even shift the marshalling up to the application. Sharing semantics are not maintained and migration is not incremental. Instead, MQSeries delivers *high performance* achieved by a combination of *asynchronous* but *guaranteed* message delivery. (It could, for example, be used as a transport protocol by an implementation of persistent spaces.) MQSeries is a well-known commercial product, widely used in industry for many years.

The application programmer interface provided by persistent spaces is simpler than MQSeries. In order to illustrate this point, an example application was downloaded from the MQSeries home page at IBM Hursley [IBM96a] (which in turn was taken from [BHL95] where the example is fully described). The example transfers the contents of a file (bytes, the simplest data type) between a sender and a receiver running on different machines.

Even without any marshalling involved, the sender program needs to call 5 procedures and uses 170 lines of C code (excluding comments). In contrast, the Library Explorer example using persistent spaces described in section 7.1.2 needs to call 3 procedures and uses 12 lines of Napier88 code (including substitution and comments). This is partly due to persistence, but also a consequence of persistent spaces and their API that were deliberately kept very simple.

6.6.2 Tuple Spaces

A *tuple space* is a repository of tuples that can be shared between two or more programs in a distributed application. A *tuple* has a key and a value, which can be an arbitrarily complex

graph. Conceptually, a tuple space behaves like a distributed object database: a program puts a tuple into a tuple space and any other program in the distributed application can then get the tuple from that space using the same key.

Tuple spaces were first proposed in the context of Linda, a set of additions to any existing programming language for supporting distributed computation. The Linda model of distribution has since then been implemented for C by the Linda Group at Yale [Fre96] and more recently for Java with GTE's WWWinda [GNSP94], the Jada research experiment [Ros96] and Sun's JavaSpaces [Wal96].

Tuple spaces have an intrinsic persistent connotation because tuples remain in the tuple space until they are explicitly removed from that space. However, actual implementations of tuple spaces are usually *not persistent*. For example, WWWinda is based on distributed shared memory and Jada uses a single, non-persistent, Java server (put and get are just remote procedure calls). JavaSpaces were designed as a support mechanism for both distribution and persistence, but at the time of writing we still have no access to implementation details.

Tuple spaces offer an IPC model very similar to persistent spaces: both permit the sharing of objects between programs by means of explicit put and get operations on a shared repository. One difference is that, while *persistent spaces expose distribution to the application*, tuple spaces behave more like a transparent distributed system.

Transparent distribution means that tuple spaces provide extreme simplicity at the cost of network delays and partial failures, against which there is little the application programmer can do. In particular, an implementation of tuple spaces based on distributed shared memory will not scale beyond the local area network. In contrast, persistent spaces amortize marshalling costs over any number of clients, keep the amount of data transmitted to a minimum, and only update the remote copies when explicitly asked by each client program.

Finally, even though current implementations of tuple spaces do not take persistence into account, tuple spaces seem highly suitable for a persistent environment. There is no fundamental reason why incremental migration and other features of persistent spaces cannot be added to the Linda model of tuple spaces. Or perhaps the opposite approach should be attempted: to integrate the best features of tuple spaces, such as extreme simplicity and flexibility, into the IPC model of persistent spaces.

6.6.3 Replication Protocols

A replication protocol maintains several copies of an object, possibly in a number of programs, all with the same value (see section 2.3.6). This is called *strict consistency* when the protocol makes an application program believe there is only a single copy of the object in the entire distributed application. There are also replication protocols based on *loose (or partial) consistency*; only these are comparable with persistent spaces because

the application programs are aware of replication and have to control consistency.

Replication protocols based on loose consistency form an extensive research area, so in this section only one typical example will be used. Lotus Notes [Lot96] is a well-known commercial product for developing distributed (collaborative) applications based on replicated documents. Notes stores documents in a server database that are copied to any number of client databases. A document can then be updated in the server or in any of the clients. From time to time, clients connect to the server to synchronize (or refresh) their replicas. Documents are very high-level objects, intended to be manipulated directly by end-users for simple replication schemes. In addition, a programming language is available for building general distributed applications based on these replicated documents. Notes can also be considered persistent, although all objects stored in the database belong to the type document. Like persistent spaces, Notes also supports incremental migration because it only transmits the updates made since the last synchronization. Notes supports refresh in both directions as opposed to only one in persistent spaces (from server to clients).

Notes is a stand-alone product with its own language and application development environment. Persistent spaces were designed and implemented as an add-on library to an existing persistent language, providing a simple interface to be used by normal application programmers. Even though it can also download code to clients, Notes is specialised on replicated documents. Persistent spaces are designed for a higher-order persistent language with a rich type system and can be used together with other mechanisms available in the language.

Finally, Notes permits each client to replicate a different set of documents. This is flexible, but it forces Notes to detect changes and marshall them in each refresh for each client. Persistent spaces can perform this marshalling locally without any information from clients, amortizing marshalling costs and reducing communication time. It could be argued that persistent spaces require the server program to explicitly tell the persistent space about all updates to the object. However, the objective is exactly the opposite: not all updates will be published to clients, only when the object reaches a stable state should the new value of the object be put into the persistent space.

6.7 Summary

This chapter proposed *persistent spaces*, a new IPC model for sharing complex, persistent objects between autonomous stores. Persistent spaces are based on a simple programming interface that lets publishers put objects into a space which can then be fetched by any number of subscribers.

The chapter includes a motivation for persistent spaces, the proposed design, interactions with other mechanisms, a detailed description of the interface to the programmer, an implementation in a persistent programming language, and a comparison with related work. The next chapter will present an example application developed to validate persistent

spaces and performance measurements.

We conclude that persistent spaces provide an effective IPC model for a certain class of distributed persistent applications, in which a server publishes objects to be used by a large number of clients. Other IPC mechanisms, such as RPC or substitution, can then be used to cover a wider spectrum of applications. This is in contrast with related work, which is either not appropriate for persistent objects (and their large transitive closures) or may not scale to many clients. When persistence is taken into account, the related work limits the complexity of the objects being shared to simple records or a single data type (e.g., documents).

Chapter 7

Evaluation

In the previous three chapters several models for building distributed persistent applications were proposed. One is a type-safe persistent RPC that passes arguments by copy. The other two, called migration by substitution and persistent spaces, are compromises that reduce the problems of passing parameters only by copy or only by reference.

This chapter describes how a real distributed persistent application was built using these three mechanisms. First, we present the code written in the application to use the mechanisms. Then we provide performance measures for both the application and other experiments we made to understand their behaviour.

7.1 Example Application

This section describes an example application that was built using the models proposed in this dissertation. The objective is twofold: to show that these models can be used by typical application programmers; and that they are worth using.

The mechanisms were utilised to build two extensions of a persistent application called the Library Explorer [Bro93, SWA+96]. The Explorer is a tool for retrieving information from the Glasgow Libraries [WWP+95]. It maintains two major data structures: *documentation* about the Glasgow Libraries (approx. 11 MB of data); and an *index* to speed up access to that documentation (approx. 500 KB).

In addition, two procedures that make use of these data structures have first to be explained (see figure 7.1).

- `searchResult` — This procedure accepts free-text *queries* specifying software components required by the user and (using information retrieval techniques) returns a list of *matches* that represent software components offering approximately that functionality.

- **`retrieveDoc`** — This procedure accepts the name of a procedure selected by the user and returns the documentation for that particular software component (typically a long string).

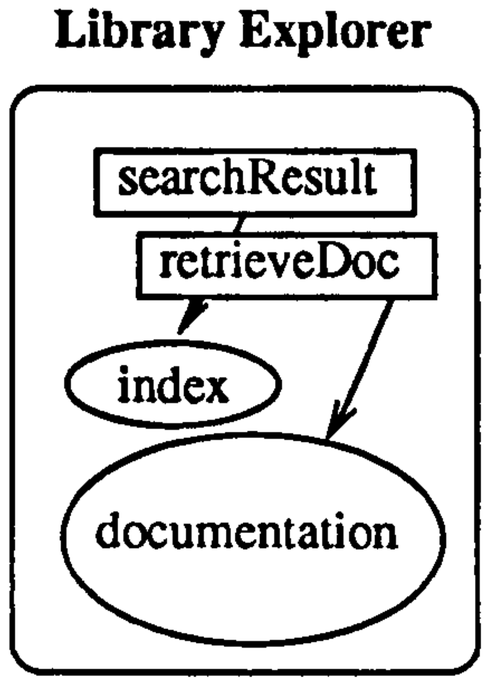

Figure 7.1: Architecture of the original Library Explorer

User interaction usually follows a well-defined pattern. First, the user writes a free-text query that is passed as an argument to **`searchResult`**, which returns a vector of name/score pairs ordered by score value. Usually, the first query is either too general (and returns too many matches) or insufficiently general (and returns no match). This is a typical problem in information retrieval.

After several interactions — adding or removing information from the query until a relatively small number of matches is returned — the user eventually selects the name of a software component. The procedure **`retrieveDoc`** is then used to obtain the documentation about that component. As a result, *documentation requests are much less frequent than free-text requests.*

7.1.1 Client/Server Explorer

In the original Explorer, the index and the documentation need to be replicated in each user's store in addition to all the Explorer code. This is a tremendous waste of store space and, as a consequence, may degrade performance. Replication also requires human and CPU time to synchronise all copies in every store with a centralized version maintained by the local Napier88 administrator.

The new client/server version of the Library Explorer permits the sharing by any number of *Client Explorers* of the code, index and documentation maintained in a single *Library Server* (see figure 7.2). The number of clients is limited in practice — depending on the Explorer usage, amongst other factors — but we hope it is large enough to take advantage of the new architecture.

This extension to the Library Explorer is implemented in a typical client/server fashion by converting **`searchResult`** and **`retrieveDoc`** into remote procedures. The *Library*

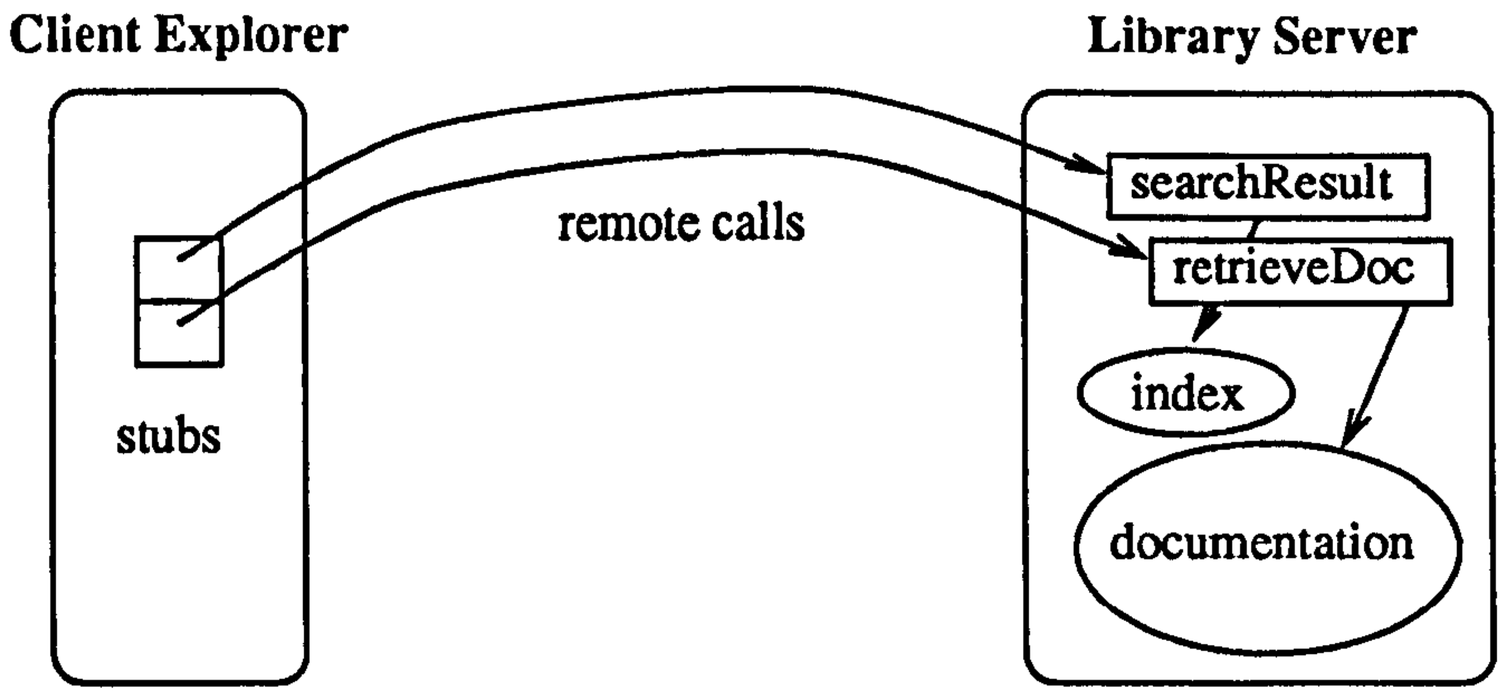

Figure 7.2: The client/server Library Explorer

Server maintains the documentation and its index about Glasgow Libraries. A light-weight *Client Explorer* running in the user store accesses the centralized Library Server using Napier/RPC to retrieve the information.

Example 7.1 shows the signatures of these procedures to give an idea of their relative simplicity, in the sense that the types of the arguments and results are either primitive types or simple data structures. (The "*" in the example represents a vector in Napier88.)

```
type resultEntry is structure( name: string; score: real )

searchResult: proc( string -> *resultEntry )

retrieveDoc: proc( string -> string )
```

Example 7.1: Remote procedures in the Client/server Explorer

Generating client and server stubs for these procedures is easy by using the programmer interface to Napier/RPC presented in examples 4.2 and 4.4 respectively.

7.1.2 Distributed Explorer

The client/server version of the Library Explorer permits the sharing of a single copy of the Explorer code and data in a server by a number of clients. However, it introduces a (slow) remote access for every user request. This separation between client and server not only slows down the Explorer (see table 7.5) but also makes the client Explorer dependent on a (remote) server, probably managed by another person. Finally, it does not scale well for many clients because there is always a single server that has to answer all requests.

The distributed version of the Library Explorer is the second extension to the original Explorer, now taking advantage of both migration by substitution *and* persistent spaces to overcome the limitations of the client/server version.

In the Distributed Explorer, the index is first published by the Library Server into a persistent space. Then it can be fetched by any number of client stores — a *Remote Explorer* in our terminology (see figure 7.3). Frequent queries to the index will now be always local to the Remote Explorer, while all accesses to the documentation for a particular software component — a less common operation — still use a remote procedure call. This is even more convenient because the documentation is also a much larger data structure than the index. The Remote Explorer is still dependent on the Library Server but only if the software component exists, was found and its documentation is requested by the user.

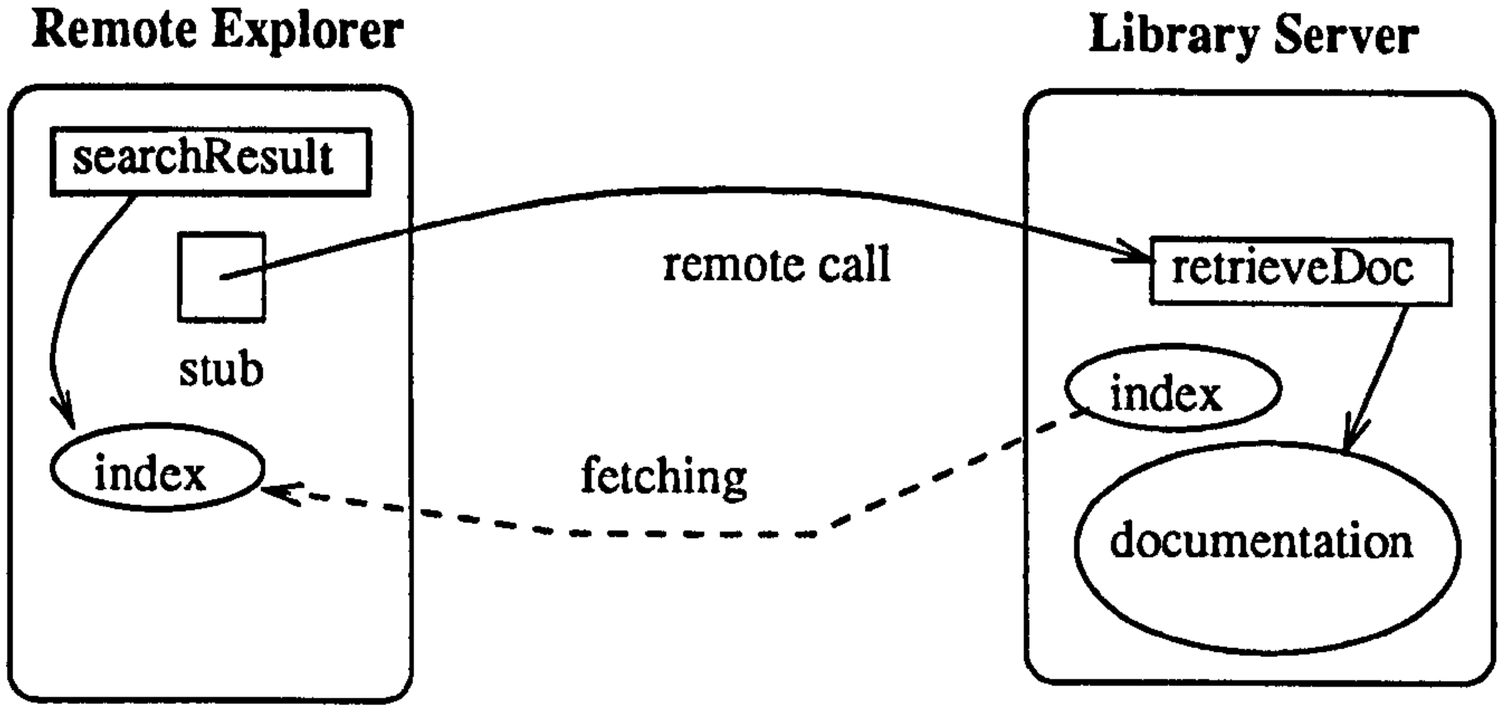

Figure 7.3: The Distributed Library Explorer

We now describe how to publish and fetch the index. The index is implemented as a *map*, a Napier88 bulk data type included in the Glasgow Libraries [ABC+93, WWP+95]. In order to create this map, the programmer needs to provide two boolean procedures over strings, `stringEqualTo` and `stringLessThan`, to permit fast access to documentation in the index (see example 7.2).

```
! define a new data type (not important in this example)
type IndexEntry is structure( count:int; weight:real; refs:*string )

! create two procedures for testing string equality and order
let stringEqualTo := proc( s1,s2:string -> bool ); s1 = s2
let stringLessThan := proc( s1,s2:string -> bool ); s1 < s2

! create an index (which is bound to these procedures)
let index := m_empty[string,IndexEntry](stringEqualTo,stringLessThan)
```

Example 7.2: Creating a `Map` instance

Objects put into a persistent space bring their entire transitive closures with them. In this case, the index would bring the two procedures `stringEqualTo` and `stringLessThan`. This is unnecessary because these procedures are well-known and unlikely to change in the future. (If they do change, then some other mechanism, such as traditional library installation, has to be used every time they change.) If these procedures were part of Glasgow Libraries, for example, then the Distributed Explorer could migrate them by

substitution when the index is published. The code to achieve the behaviour just described is presented in example 7.3.

```
! creates a new persistent space
let explorer = publishSpace( "Explorer" )

! declare the procedures as substitutable
! the values will be accessed by following the path
substitute( explorer, "/stringEqualTo" )
substitute( explorer, "/stringLessThan" )

! puts the index in the space but not the procedures
putObject( explorer, "exploreridx", any(index) )
```

Example 7.3: Publishing the explorer index

After publishing the persistent space and putting the index into that space, any client store running the Remote Explorer can then subscribe to the space and fetch the index (see example 7.4).

```
! creates a handle for the persistent space
! the name space for "Explorer" is the server
let explorer = subscribeSpace( server, "Explorer" )

! declare the procedures to replace the surrogates
! the values will be accessed by following the path
replace( explorer, "/stringEqualTo" )
replace( explorer, "/stringLessThan" )

! copies the space to the local store
! synchronization between replicas occurs here
fetchSpace( explorer )

! unmarshals the space in the local store
buildSpace( explorer )

! makes the index visible to the target program
let index = getObject( explorer, "exploreridx" )
```

Example 7.4: Subscribing to the explorer index

Two of the main advantages of using a persistent space in the Distributed Explorer are flexibility and autonomy. The subscribers decide when to fetch new versions of the index, allowing them to choose between using a slightly out-of-date index and the very latest version. In addition, each subscriber has its own version of the index independently of

other subscribers or the publisher. Similarly, the publisher can be constructing a new version autonomously and then choose when to publish it and make it available.

The Distributed Explorer may pay off depending on the time for the remote requests, the time to migrate the index, and the number of accesses to the index before it is replaced with a more up-to-date version (see section 7.2). It could be argued, however, that this library example is particularly well-suited to this mechanism.

One of the goals of substitution and persistent spaces is that these models should be easily understood and used by application programmers. The relatively few lines of Napier88 presented in examples 7.3 and 7.4 show the ease with which they can be used to introduce distribution into an existing persistent application that has more than 11,000 lines. The amount of extra code and its complexity is relatively small compared with the source code for the entire application. Only localized changes are required to introduce distribution and these do not disturb the rest of the application.

7.2 Performance Measurements

In this section we present, analyse and draw conclusions from preliminary performance measurements using the mechanisms proposed. We have conducted these experiments using both the extensions to the Library Explorer described in the previous section and programs written specifically to understand particular aspects of the implementation.

The goal of this section is to show that the implementation provides *acceptable performance* when compared with Napier88, the original Library Explorer and similar mechanisms in other programming languages.

Methodology

We aim to measure the behaviour of systems and applications under *typical working conditions.* For this reason, all measurements in this section are taken "warm", i.e., after all set-ups have been made, indexes built, and caches primed.

The measurements do not include hand-crafted optimisations based on knowledge of the implementation of Napier88 or the distribution mechanisms themselves. For example, in the Explorer's case the measurements use code written by one of the implementors of the Explorer, Stewart Macneill.

There are also other factors—such as threads and garbage collection—that introduce variability and thus some inconsistency in the results. Even though we could control these factors to some extent, we do not want to do it because *they are part of the Napier88 system* and to make such changes would result in atypical (or artificial) comparisons.

- *Threads*— Napier88 threads are used in Napier/RPC for building the time-out mechanism when the server does not respond. Threads are implemented by the run-time system itself and the scheduler is based on "a fixed time-slice by the number of instructions" [Mun93]. Performance of one thread in Napier88 is thus inversely proportional to the number of threads running in the (single processor) system and this significantly affects any performance measurement.

- *Garbage collection*— The Napier88 run-time system has both a memory and a disk garbage collector [Bro88, BR90, Mun93]. The memory garbage collector may start at any time and is based on the "stop the world" principle. It is therefore likely to affect *long* measurements.

The two machines used for the experiments are the same DEC Alphas with OSF/2 (also called Digital UNIX) that are normally used to run Napier88, connected via the departmental 10 Mb/s Ethernet. These machines have enough main memory (more than 64 MB) to work with our store sizes, which are also typical of small Napier88 stores (between 50 and 100 MB). These experiments were run after the normal working hours but the network and the machines still had a small number of users that also interfered with the measurements.

For all these reasons, the numbers presented in this section can only be considered very crude measurements of the mechanisms proposed and their implementation. Nevertheless, they portray some useful information.

7.2.1 Remote Procedure Call

The performance of the basic RPC mechanism is presented in this section. We start with measurements of the absolute performance for a minimal remote procedure call using Napier/RPC, followed by a detailed analysis to check where the time is being spent. We also present measurements for the client/server version of the Library Explorer described in section 7.1.1 above.

Absolute Performance of a Minimal RPC

Table 7.1 shows an experiment with *minimal procedure calls*— that is, with the simplest types (i.e., integer) as arguments and result— intended to measure the absolute performance of remote calls using Napier/RPC. The numbers in the table are reported by the operating system in "hardware-dependent clock ticks" and are the average for at least 100 repetitions. *User CPU* means the time spent executing the operation in the user address space, *O/S CPU* is the time spent in the operating system, and *Elapsed Time* the wall-clock time. Only the three most significant digits are presented.

From table 7.1 it is clear that the absolute performance of Napier/RPC— 2.5 seconds for a remote call— is clearly problematic. (The numbers in each row do not add-up because of other processes running on the same machine and the time spent in the network and

	Time		
Procedure	**User CPU**	**O/S CPU**	**Elapsed time**
Local (microseconds)	38.8	2.47	58.6
Remote (seconds)	1.61	0.198	2.50
Ratio	*41,500*	*80,000*	*43,000*

Table 7.1: Performance of Napier/RPC

with remote execution at the server.) The table also shows that Napier/RPC is 5 orders of magnitude slower than a local procedure call in Napier88.

The bottleneck resides in one of the following areas: 1) *packing and un-packing* caused by this particular implementation of Napier88 or the fact that Napier/RPC itself has not been optimised; or 2) *data transmission* as a result of Napier88, Napier/RPC or the underlying system (O/S and network costs).

However, there are two reasons to believe that data transmission is *not* the limiting factor: the marshalled argument plus control information (for type-checking and so on) only requires 108 bytes to be sent to the server program; and the table shows that the time spent at the operating system is not significant. It is interesting to go further and measure the time spent just on data transmission.

Time Spent on Data Transmission

We will now check if data transmission represents a significant cost for minimal calls by measuring the time spent transmitting data as raw bytes between stores.

In the current Napier/RPC implementation, byte arrays of 4 bytes each are transmitted until the 108 bytes representing the minimal remote call (see above) are sent. Table 7.2 shows how the time needed to transmit the 108 bytes varies depending on whether 4 or 108 bytes are used in each byte array — that is, 27 transmissions if 4 bytes are used or a single transmission if 108 bytes are used. The *ratio* between the two options is also presented. (The numbers in the table are an average for at least 100 repetitions.)

	Time		
Array size	**User CPU**	**O/S CPU**	**Elapsed time**
4 bytes (seconds)	0.00330	0.00160	0.00587
108 bytes (seconds)	0.000451	0.000113	0.000664
Ratio	*7.32*	*14.2*	*8.84*

Table 7.2: Time to transmit 108 bytes in Napier88

The table shows that using byte arrays with 4 bytes is approximately 9 times slower than the alternative of sending a single byte array with 108 bytes. In any case, *the time spent transmitting the data is a negligible part of the total elapsed time for a minimal remote call* — compare the 0.00587 seconds in table 7.2 with the 2.5 seconds in table 7.1. We

conclude that packing and un-packing is the dominant activity in a minimal remote call for Napier/RPC. (It is also dominant with large arguments, see table 7.8.)

Relative Performance of Remote and Local Calls

Napier/RPC is 5 orders of magnitude slower than a local procedure call in Napier88 and limited by packing/un-packing. Both results may well be a consequence of Napier88, the language in which Napier/RPC is implemented. In order to abstract from the particular implementation of the language, we now compare the *relative performance* of Napier/RPC—the *ratio* between remote and local calls—with another RPC system.

This comparison uses only the relative performance and thus should indicate whether it is the Napier/RPC implementation that is particularly inefficient or Napier88 itself. The test is important for a number of reasons: 1) application programmers, having made a decision to use a particular language, then have expectations mainly in the context of that programming environment; 2) Napier88 could be implemented with performance similar to other languages; and 3) the mechanisms under investigation could be built below the language level or for languages other than Napier88.

Table 7.3 below presents measurements for minimal remote calls using Sun/RPC [Sun93b] and *equivalent* local calls in C. The idea is to compare these numbers with those presented in table 7.1 for Napier/RPC and Napier88 respectively.

	Time		
Procedure	**User CPU**	**O/S CPU**	**Elapsed time**
Local (nanoseconds)	143	zero	143
Remote (milliseconds)	0.0623	0.20	1.76
Ratio	*436*	—	*12,300*

Table 7.3: Performance of Sun/RPC

The table shows that a C program can make more than 12 thousand minimal local calls in the time taken to do one minimal remote call using Sun/RPC, whereas table 7.1 shows that Napier88 could make 43 thousand when compared with Napier/RPC. (The "zero" in the table for the time spent with O/S CPU during a local procedure call actually represents a number so small that it may be considered negligible.) *If we normalise for language speed, than Napier/RPC is only 3.5 times slower than Sun/RPC.* The difference may be the cost of type-checking in Napier/RPC, the investment in optimisation in Sun/RPC or transmission costs.

Table 7.3 also shows that the time spent with *O/S CPU* dominates a minimal remote call in Sun/RPC. This is a good indication that Sun/RPC is limited by transmission costs since marshalling does not consume O/S CPU. In Napier/RPC the bottleneck is the time spent in the user space, mostly packing and un-packing (there is nothing else CPU intensive going on at call time).

Another interesting comparison is the *ratio* between the time spent in *User CPU* for remote and local procedure calls in Napier/RPC (41,500) and Sun/RPC (436). These numbers mean that the Napier/RPC implementation is a very inefficient consumer of CPU compared with Sun/RPC, although part of the problem is caused indirectly by the Napier88 implementation.

Comparison with Another Persistent RPC

There is a major difference between C and Napier88: while C is compiled directly into machine code, Napier88 is compiled into byte code and interpreted. Thus we believe it is useful to compare Napier/RPC with another RPC for an interpreted language, especially another *persistent* language.

For this experiment we chose Tycoon/RPC (see below) because it is the only persistent RPC system for which we know of recently published performance measurements.

Table 7.4 below shows the time needed to perform local and remote procedure calls for three different RPC systems in three programming languages. The numbers for Sun/RPC and Napier/RPC are taken from tables presented earlier in this section. The numbers for Tycoon/RPC are taken from [Mat96] and are meant to represent only *approximate values* since they are based on similar but different conditions (Tycoon on PCs, also connected by an Ethernet LAN, but running Linux). For this reason, in this table we are only interested in comparing the *ratios* between remote and local calls for each RPC system, not their absolute performance and even less that of their host languages.

	Elapsed Time		
Procedure	**Sun/RPC**	**Tycoon/RPC**	**Napier/RPC**
Local (microseconds)	0.143	2	58
Remote (milliseconds)	1.7	130	2,500
Ratio	*12,000*	*65,000*	*43,000*

Table 7.4: Comparison of *ratios* for local/remote calls

Table 7.4 shows that all three RPC systems present the same order of magnitude when the relative performance of remote calls is compared with local calls (that is, tens of thousands). If we assume that Tycoon/RPC does not use exorbitant amounts of data being transmitted in a minimal remote call, then it can be concluded that *packing and un-packing becomes the limiting factor for RPC performance in a persistent language.* On the other hand, for simple compiled languages like C the network is the bottleneck. It remains a challenge to find out why this happens and achieve the same result in a type-safe persistent programming language.

Library Explorer

We now present measurements for the basic Napier/RPC performing under the client/server version of the Library Explorer described in section 7.1.1.

Table 7.5 shows the time spent during three different user requests for software. The number of matches is limited to the first 10 ordered by score to limit the amount of data sent back from the server (the total number of matches is shown in parenthesis). Each request was repeated 10 times for each of the local and remote versions of the Explorer. The elapsed time is reported as the *minimum* and *maximum* times achieved. The granularity is one second.

Task Requested		Elapsed time (seconds)	
Free-text Query	Matches	Original Explorer	Client/Server
"View slides"	1 (1)	1–1	3–8
"Display an image"	10 (55)	1–2	5–10
"Write a string"	10 (279)	2–3	6–11

Table 7.5: Performance of the Client/server Explorer

The numbers reported for the client/server Explorer can be interpreted as being the sum of the component times required for every remote procedure call.

1. A constant time for a minimal remote call (2.5 seconds, see table 7.1).
2. The time to pack and un-pack the string representing the query minus the time to pack and un-pack a minimal argument (a short time in this case, since the argument is just a few characters).
3. The time to transmit the packed argument (negligible).
4. The processing time of the remote procedure itself (1 to 3 seconds).
5. The time to pack and un-pack the matches minus the time to pack and un-pack a minimal result. This time increases quickly with the volume of data representing the result value (see section 7.2.3) and may be responsible for most of the time difference between the original and the client/server versions of the Explorer.
6. The time to transmit the packed result (negligible).

Performance can sometimes divert application programmers from an important semantic difference between the original Explorer and its client/server version. This difference concerns the fact that *copies* are passed as arguments and results in a remote call, as opposed by passing these by *reference* locally in the original Explorer. In the Library Explorer this difference does not create any problems because the result value is just displayed for a very short period of time (compared with the rate of updates in the index itself).

7.2.2 Migration by Substitution

Performance measurements for migration by substitution are now presented in this section. However, the reader should always remember that the major contribution of substitution is semantic—it allows objects that refer to non-migratable parts of the store to migrate (see section 5.4). The question here is whether substitution can perform this work well enough to be useful in real applications.

Simple Experiment with Substitution

Table 7.6 shows an experiment conducted to compare the transfer times of a small data structure with and without substitution. The data structure transferred is an empty `Map[int,string]`, a standard Glasgow Libraries map with integer `equalTo` and `lessThan` tests (similar to the Explorer index, see section 7.1.2). The numbers in the table represent an average over 10 migrations.

Procedures being...	Data transferred		Elapsed time (seconds)
	Objects	Bytes	
transferred	15	404	13
substituted	8	300	7

Table 7.6: Performance of migration by substitution

The numbers show that the amount of data transmitted to migrate these two procedures is small. In addition, the absolute time taken to transfer an empty data structure with two procedures (13 seconds) does not seem exaggerated when compared with the time needed to perform a minimal remote call (2.5 seconds).

Despite that, a substantial gain in performance is achieved by substituting the two procedures. This gain is mainly for two reasons.

1. Not transmitting these procedures substantially reduces the time for packing and un-packing their values because the surrogates that represent the procedures for substitution are very small.

2. In the current implementation of Napier/RPC, the procedures are transmitted as hyper-programs (see section 3.2.4). Avoiding compilation of the hyper-programs at the target probably reduces the transmission time much further since compilation is an extremely expensive operation (see section 5.5). (On the other hand, in a system that transmits byte code, such as Java, the saving will be proportionally less.)

Other researchers have presented numbers within the same order of magnitude for transmitting code between programs. Examples include the 5–45 seconds for migrating small to medium size applications in Obliq [BC95b] and up to a few seconds for procedures in Facile [Kna95]. Downloading Java *applets* [AG96] can also take several seconds even when bandwidth is available, e.g., the example application in the MarketPage Web site [Bul96].

7.2.3 Persistent Spaces

A number of performance experiments with persistent spaces are now presented. We measured both purpose-built programs to check their incremental and scalable behaviours and the Distributed Explorer presented in section 7.1.2.

Incrementality of Persistent Spaces

Table 7.7 presents six experiments that put (pack) the updated value of a `Map[int, string]` (similar to the one presented in the previous section) into a persistent space and build (un-pack) the map in a client program. The granularity is one second. The time taken to fetch the map to the client is not presented to concentrate on the dominant activity of persistent spaces. The two procedures that belong to the *map* (for integer equality and comparison) now always migrate by substitution.

Exp Nb	Value transmitted	Data transferred		Time (seconds)	
		Bytes	Objects	Packing	Un-packing
1	Map with 10 entries	3,144	93	2	3
2	Exactly the same map	164	4	1	2
3	Another 10 entries	1,400	37	2	3
4	Another 20 entries	2,372	67	3	4
5	Another 40 entries	4,468	127	7	6
6	Another 100 entries	10,240	307	16	12

Table 7.7: Incrementality of persistent spaces

Measurement 1 (first row) shows the initial cost to migrate the map with 10 entries. Measurement 2 migrates exactly the same value to check the performance of a "null update".

The numbers show how persistent spaces use *incremental migration* to reduce the amount of data being transferred if large parts of common data structures remain unchanged between migrations. In this case the number of objects come down from 93 to only 4 objects. (It could be possible to reduce this number further by optimising the marshalling and unmarshalling algorithms.)

Interpretation of measurements numbered 3 to 6 requires understanding of an aspect of the map implementation. A map contains organisational data, which varies only slowly as entries are added. In this case the organisational data was about 2,000 bytes for a small map. It then has data representing the entries, in this case about 100–130 bytes per entry. Taking this into account, *the amount of bytes transferred is proportional to the volume of new data published each time* (plus any eventual changes to the organisational data).

The time to pack and un-pack the data structure seems proportional to the numbers of objects being transferred. However, a map with 100 entries cannot be considered a large data structure, so below we check if this is the case for a larger number of objects.

Scalability of Persistent Spaces

We now analyse how the time for packing and un-packing large data structures grows compared with the number of objects being transmitted. According to our earlier definition of scalability (see section 2.3.5) we expect the time to grow *linearly* with the number of objects if the algorithms are *scalable*.

Figure 7.4 is a graphical representation of the time needed to pack and un-pack the same *map* above by increasing the number of objects. We start with 200 entries that translate into 671 objects. The largest map transferred in this experiment contains 2,000 entries or 6,115 objects. (In order to have a clearer representation of the overall trend and be able to generalize for even larger data structures, we kept the number of running threads small and constant for this particular experiment. We also switched off the time-out facility for fault-tolerance.)

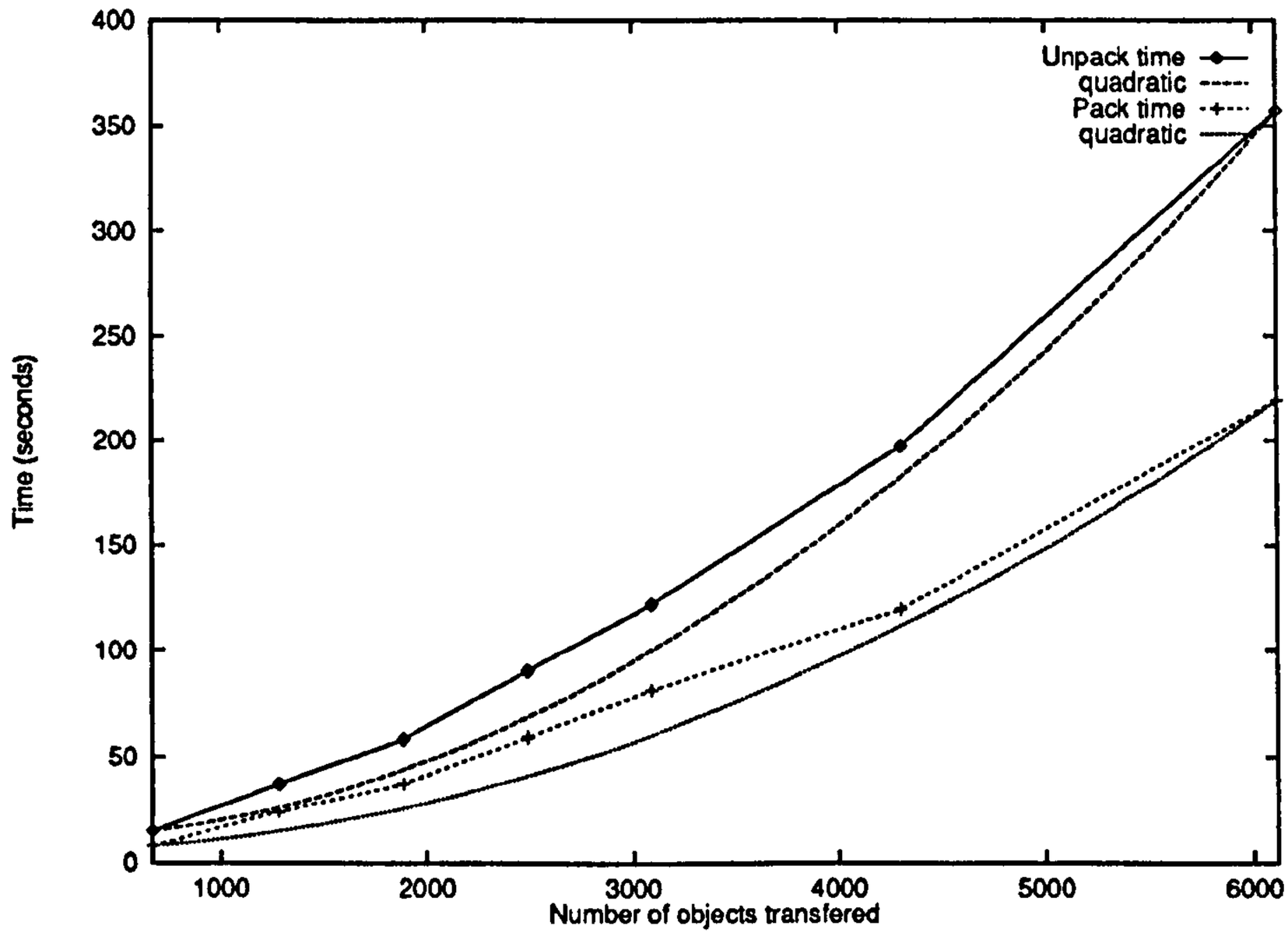

Figure 7.4: Scalability of persistent spaces

The figure shows that *packing is faster than un-packing for large data structures.* A possible explanation for this behaviour is the space that has to be allocated in the store to build the objects during unmarshalling.

The figure also shows that the time needed for both packing and un-packing increases very close to N^2, where N is the number of objects transmitted. This is a natural consequence of the algorithm that requires a *sequential scan* unless there is a *unique* (hashable or sortable) object identifier that is *stable* during (un-)marshalling. If a compacting garbage collector is running, it is difficult to have access to such an identifier at the language level.

A time proportional to N log(N) can be achieved by re-implementating persistent spaces

with low-level access to the run-time system. There are two approaches to implement this unique, stable identifier: 1) every object has a *persistent identifier* that remains constant for the duration of the space; or 2) use memory addresses and re-hash all auxiliary data structures after a garbage collection. (Tycoon/RPC [Mat96] is based on the first approach.)

Distributed Library Explorer

We now measure the Distributed Library Explorer presented in section 7.1.2. The index of the Library Server used for this experiment has 3,389 entries and corresponds approximately to 500 KB of data.

Table 7.8 shows measurements for creating, packing, transmitting and un-packing the index. The operation *Create Index* (part of the original Explorer) and *Put Index* (into the persistent space) are *local* to the Library Server. The *Build Index* operation is *local* to the Client Explorer. Only the *Fetch Space* operation actually needs a remote connection but just transmits bytes.

Task Requested	Data transferred		Elapsed time	Place of Operation
	Objects	Bytes		
Create Index	—	—	38 minutes	On Library Server
Put Index	10,486	—	18 minutes	On Library Server
Fetch Space	—	500 KB	4 seconds	Remote Operation
Build Index	10,486	—	37 minutes	On Client Explorer
Typical Query	—	—	1–3 seconds	On Client Explorer

Table 7.8: Performance of the Distributed Explorer

The numbers in this table are consistent with those presented above in figure 7.4 since for large data structures the time to un-pack (build) exceeds the time to pack (put).

The advantage of caching the index at the Client Explorer outweighs the cost of transmitting it only if the index remains stable for a period greater than a few hundred queries. Unfortunately, our limited experience with the Distributed Explorer is not sufficient to know if this assumption is realistic. In any case, end-users can always compromise on the consistency of the index to avoid paying the price of a remote fetch every time the index is updated. Moreover, as a result of incrementality, only the new parts of the index since the last migration are actually fetched and un-packed.

Finally, the 4 seconds of connection time in the table are specially interesting because they reduce dependency between publisher and subscriber to a minimum. This minimum—which we call the *minimal connection time*—makes packing and un-packing local to avoid dependencies on another store for long periods of time. It may also prove useful in mobile computing and other environments where bandwidth is limited, unreliable or expensive.

7.3 Summary

This chapter described an example application for the mechanisms we proposed. The example is based on extensions to an existing persistent application called the Library Explorer.

First, a client/server version of the Explorer was presented that makes use of the basic RPC mechanism. Then we presented the Distributed Explorer, a second extension that caches the index in the client to perform most of the queries locally. As our mechanisms were useful for building these extensions to the Library Explorer, then they may also be useful for building other distributed persistent applications.

We then presented and analysed performance measurements for both purpose-built experiments and in the context of the extensions to the Library Explorer.

The performance of the current implementation of Napier/RPC is disappointing. For example, a minimal remote call takes 2.5 seconds while Sun/RPC is almost 1,000 times faster. However, Napier/RPC has approximately the same *relative performance* as Sun/RPC (see table 7.4) if we take into account the performance of Napier88 and C respectively, in which applications using these mechanisms are built. A factor of 3.5 in relative performance (43,000 divided by 12,000) remains to be explained, but this may be the cost of type-checking, lack of optimisation or transmission costs.

We also measured substitution and persistent spaces. A preliminary experiment with migration by substitution suggests it may substantially improve performance by not migrating procedures but by substituting them. Finally, measurements made with persistent spaces indicate that our packing and un-packing algorithms are proportional to N^2, where N is the number of objects transmitted. The cause of this was identified and a sketch was given of ways to reduce this time to an acceptable N log(N) behaviour (see also section 8.2.2).

On the other hand, a major feature of persistent spaces is their support for minimal connection time based on the separation between the long time required for packing and un-packing and the (relatively) short time needed for data transmission. Incremental migration has also presented promising performance.

We conclude that the models proposed in this thesis are: simple to understand and use, useful for building certain distributed persistent applications; and may have eventually an acceptable performance compared with Napier88 and other RPC systems—since there is nothing fundamental preventing an efficient implementation of these models at the run-time system level (see section 8.2.2).

Chapter 8

Conclusion

In this dissertation three models for higher-order, type-safe, distributed computation over autonomous persistent object stores have been proposed and implemented. We have also presented in chapter 7 how we used and measured them to validate their effectiveness in building real distributed persistent applications (at least as a "proof of concept").

This chapter presents a summary of the thesis, lists some possible research issues for future work, and compares the goals achieved with the initial IPC design issues described in chapter 2. Throughout the chapter, the reader should always remember the thesis statement presented in section 1.2: design, build, use and test an IPC mechanism for a persistent environment that is simple, general and realistic.

8.1 Summary of the Dissertation

Three IPC models were proposed in this thesis, implemented and used for building an example application. In addition, a preliminary investigation of their usage and performance was made. In this section, we point out the most important contributions of each model followed by a joint description of usage experience.

8.1.1 Models Proposed

Here we briefly describe the models proposed in this thesis, giving particular emphasis to the major features and limitations of each model: type-safe persistent RPC; migration by substitution; and persistent spaces.

Type-safe Persistent RPC

Our first IPC model, described in chapter 4, has two important features when compared with more traditional RPC mechanisms.

1. Napier/RPC is *strongly type-safe.* Furthermore, type-safety is achieved without requiring any extra effort from application programmers when compared with other non-safe RPC systems.
2. The Napier/RPC implementation *takes advantage of reflection and other features of Napier88* to provide novel mechanisms such as run-time stub generation. As a result, a simpler interface than other RPC systems can be presented to application programmers.

It is very important to stress that the RPC mechanism was entirely built in Napier88; the language was not extended nor was a separate language needed. As a consequence, the RPC implementor was supported throughout the implementation work by type-safety and other Napier88 features (see section 3.2.1) available to all persistent application programmers.

Napier88 also provided features not commonly available in other programming languages that helped to implement the RPC. For example, Napier88 permits access to type information at run-time and already supports (structural) equivalence comparison between two types defined *independently.* As a result, strong type-safety can be enforced with minimum effort, based on existing Napier88 technology.

Napier/RPC has a very simple interface. Programmers no longer have to write descriptions of procedures in a different language such as IDL [BN84, Sun93b, OMG95, Sun96b]. Napier/RPC simply generates the client and server stubs from within the language during normal program execution. Using run-time reflection, Napier/RPC discovers the signature, generates the code for the stub, compiles it and then uses persistence and first-class procedures to install it in the store. Type-safe dynamic binding allows these compiled stubs to be re-used at a later time.

It should be noted that it is also possible to avoid the need for a separate language like IDL with a conventional programming language. For example, keywords or other language mechanisms can be used to identify the remote procedures and the compiler changed to generate the stubs automatically [KOMM93, KKM94]. However, with a language like C this requires access to the compiler source code and significant programming effort, while in Napier88 this is possible by writing a conventional application in the standard language.

The key contribution was the demonstration that type-safety can be extended to the IPC mechanism without requiring any extra effort from the application programmer. Another contribution is the use of persistent features to support run-time stub generation within the language itself.

Migration by Substitution

The basic Napier/RPC described in chapter 4 was still very restricted in the range of types supported as arguments to remote procedures. For example, procedures cannot be passed as arguments, although they are first-class values in Napier88. This restriction is imposed not only because marshalling procedures is complicated — a problem that was solved with more work — but especially because *procedures typically have large transitive closures* that make eagerly passing arguments by copy inappropriate.

This limitation on argument types is not restricted to Napier88 or even to persistent programming languages. The same problem arises in traditional languages that support first-class procedures and other complex data types. However, persistence *amplifies* the problem since the transitive closure of a procedure may include large collections of objects in the stable store.

The problem with transitive closures can be avoided by restricting the class of migrating procedures to *self-contained procedures* (those without references to other objects) or *standard procedures* (those guaranteed to exist in every application environment so that copying them can always be avoided). For example, Java applets [AG96] have to inherit from a special class called *Applet* and preferably should use only Sun's standard classes, otherwise they cannot be transmitted or may fail remotely.

Migration by substitution is a first attempt to solve the much more difficult, but also more rewarding, general case. Using simple primitives, application programmers can define procedures and other objects to be substituted by surrogates just before migration. On arrival at the target, these surrogates are replaced by local versions of the objects substituted. Substitution then limits the transitive closure of objects reached from procedures and makes marshalling them both feasible and worthwhile.

The mechanism guarantees that *the local and remote copies have identical type signatures* since these have to be checked before any migration. However, substitution does not compare their behaviour. Not only is the comparison difficult to perform, but this freedom also lets programmers specialise objects to take advantage of local conditions.

Using substitution, we have successfully migrated a number of complex procedures not possible before by declaring as substitutable all objects belonging to the Napier88 Standard Library, the Glasgow Libraries and other libraries specific to the application. Free variables are still copied to the target store, so procedures may carry with them data and other procedures if needed.

The key contribution is that large objects, or objects that refer to large objects, can now (virtually) migrate without compromising too much on store autonomy. Substitution only needs coordination between the source and target stores at two well-defined periods of time: for checking the types of the substitutable objects (only once per long-lived session) and for migrating the objects themselves. After migration, each store can proceed autonomously as with normal RPC.

Persistent Spaces

Although substitution helps to solve a major problem with migration by copy in a persistent environment, the basic parameter passing semantics is still by copy, i.e., duplicating the value of the arguments remotely. Copy semantics works perfectly well for immutable types (like integer and string) and may be acceptable for small, mutable data structures.

The remaining problem is that large, evolving data structures pose new challenges solved neither by copying nor by substitution.

1. How to access remote objects without remote references ?
2. Once a copy is made, how to maintain coherence between the replica and the original object ?
3. If part of the object has already migrated, how to rebuild the original sharing relationships in the target store ?

Persistent spaces are an attempt to answer all these questions in an integrated manner. A persistent space is conceptually a repository of objects into which a publisher store can put objects and from which any number of subscriber stores can retrieve copies of these objects.

As the name indicates, persistent spaces use persistence to *remember the values and relationships between objects and their sub-objects* (object components). If a sub-object of a large object has already been put into a persistent space, then *only the new sub-objects are copied to the space.* Furthermore, the relationship between the new sub-objects and other objects already in the space is maintained.

Persistent spaces have a number of other advantages compared with plain RPC, one of the most interesting being incremental migration. By incremental we mean that large objects do not need to be put into a persistent space entirely at once.

The key contributions of persistent spaces can now be described just by answering the three questions above.

1. Remote objects can always be accessed directly by remote procedure call, but this is a slow and unreliable operation. Persistent spaces provide a more flexible mechanism to share the values of public objects with many other stores by creating remote copies. They amortize the cost of marshalling, reduce coupling between client and server, and provide a naming context.
2. Persistent spaces can also be used to propagate new values to remote stores, and efficiently because only the difference with the previous version is copied.
3. The sharing relationships are rebuilt remotely by re-using (instead of re-transmitting) objects that are part of another (larger) object, most of which has not changed since the previous migration.

It is of interest to note here that JavaSpaces [Wal96]—proposed by Sun in the context of Java [AG96]—are similar to persistent spaces. Based on Linda [Fre96], JavaSpaces allow a publisher to put tuples into a "space" similar to a persistent space. These spaces maintain serialized (marshalled) values that may then be accessed by many clients repeatedly or explicitly removed. The main difference from persistent spaces is that *JavaSpaces and the publisher are separate* so that many publishers may use the same space.

Marimba, a new company formed by members of the original Java team, has very recently announced a new product called Castanet [Mar96]. Castanet proposes *channels* to distribute code (Java *applets* or full applications) and data (for example, the contents of a Web site). A channel can be subscribed to and downloaded, and is automatically updated. Like persistent spaces, channels cache code and data locally to avoid remote calls. There is also a single publisher and many subscribers. Unlike persistent spaces, a subscriber always has the last version of the contents of a channel it has subscribed to.

8.1.2 Usage Experience

In order to test the models proposed, we have implemented and used them in a number of applications. In this section we present a summary of one of those applications and the main conclusions regarding utility and performance.

Example Application

The example application for our proposed models extends the Library Explorer as described in chapter 7.

First, we used Napier/RPC for building a client/server version of the Explorer with all user requests performing a remote procedure call. The simple interface made Napier/RPC easy to use for the Explorer programmer.

In the second extension, called Distributed Explorer, we used both migration by substitution and persistent spaces to cache the Explorer *index* at the client. As a result, most queries now run locally without the need for a remote call.

The index is a complex and quite large data structure (500 KB) that includes two procedures. Using migration by substitution we avoid copying these procedures, thus accelerating migration of the index, since copying procedures is particularly expensive. Persistent spaces permit the index to be updated incrementally, with only the new parts of the index having to be copied and re-built at the clients.

Performance Measurements

The current implementation of all three mechanisms is unacceptably slow. For example, the basic type-safe RPC takes 2.5 seconds for a minimal remote call.

These performance numbers are mainly a consequence of implementing Napier/RPC entirely in Napier88. In order to abstract the performance of our mechanisms from the implementation language, we also measured remote calls in other RPC systems and local calls in their implementation languages. These measurements indicate that the *relative* performance of Napier/RPC—that is, the *ratio* between remote calls when compared with local procedure calls—is of the same order of magnitude as other RPC systems.

The preliminary measurements we made using migration by substitution indicate that a dramatic performance increase can be achieved by avoiding the migration of large data structures. However, the most important contribution of substitution is that objects containing references to very large or immobile objects (that already exist remotely) can now migrate.

We also measured persistent spaces. It takes a few seconds to transmit a small data structure and minutes to migrate an Explorer index with 10,000 objects containing 500 KB of data.

These numbers are unacceptable. However, we have to take into consideration that Napier/RPC is implemented in Napier88, an interpreted persistent programming language 3 orders of magnitude slower than C (see tables 7.1 and 7.3). It is probable that re-implementation of these mechanisms within the Napier88 run-time system (i.e., in C) would achieve an adequate performance. Similarly, re-implementation of these mechanisms for some compiled strongly-typed language should also achieve acceptable performance.

More worrying is the time for packing and un-packing that grows quadratically with the number of objects. The non-linearity of Napier/RPC can be explained by not having access to a unique and stable object identifier at the Napier88 level. (This results in a linear search to find out if an object has already been packed.) However, the time for (un-)packing can be made proportional to N log(N) in the run-time system itself or at the language level with low-level access to the run-time system (see *Scalability of Persistent Spaces* in section 7.2.3).

Summary

The section presented our experience and that of other programmers with the models proposed. The models and their respective implementations were simple enough to be understood and useful for building a practical distributed persistent application.

There are problems with performance in the current implementation, but we have sketched ways of solving it in section 7.2.3 and will describe these better in section 8.2.2.

Overall, the initial goal of delivering programming models more adequate for building a particular class of distributed persistent applications has been achieved.

8.2 Future Work

The dissertation has demonstrated that applications can be built which make use of both persistence and distribution. However, the combination of these two (typically disjoint) worlds is only possible by making a number of compromises. We have addressed three specific areas where these compromises are most needed: higher-order migration, type-safe computation, and autonomy between stores.

In the future this research can be taken in several directions, including: mobile object systems; implementation issues; distributed persistent applications; and heterogeneity and inter-operability.

8.2.1 Mobile Object Systems

A mobile object — also called network or mobile agent — is an active object that can migrate between processes. Mobile object systems are run-time systems that support mobile objects [BTV96]. By "active" we mean these objects are not composed solely of data but should include code as well.

Mobile objects can be seen as representing a particular class of distributed persistent application but they are becoming ever more important with the rise of the Internet — and its slow, unreliable connections. They are now considered a separate research area; the same is happening, for example, with digital libraries.

The research area of mobile objects shares many of the same topics developed within the scope of this thesis, although addressing a broader range of issues. Below we give some examples of crucial research issues.

- Mobile objects should have "independent behaviour" (otherwise they would be just like data structures) so *migrating code* is a major issue in object mobility. Should the code be pre-installed in a central repository as in MOLE [SBH96], migrate on-demand like Telescript [Whi94b, Whi94a] and Java [AG96], or be remotely executed as Kato and others propose [KMK96, KTM+96] ?

- Mobile objects refer to other objects, so there is always the problem of what to do with *free variables* and references to the local environment. Should they be copied or passed by reference ? If copied, should any replication protocol be enforced ? Should they be automatically duplicated as proposed for Tycoon [MMS96] ?

- Mobile objects execute remotely, so *security issues* are extremely important. Should

agents have limited functionality like *applets* in Java [AG96] ? Should they execute in a separate address space (from the host system) or even in another computer ?

- Mobile objects need to contact and access other objects, so *communication* is an important piece of the puzzle. How mobile objects indicate whether interaction is with the local objects or with those on a "home" site ? Should mobile objects use a name server, dynamic binding or static binding ? Should they be free to follow references in the local store or should they have a well-defined, restricted "access list" of objects they can talk to ?

Programming systems like Telescript [Whi94b, Whi94a] and Java — with its *applets* [AG96] and *servlets* [Sun96d] — are early examples of the technology that will support these mobile objects. Telescript was specifically designed as an agent language; while in the beginning it was a proprietary environment, it has recently changed its focus to the Internet. Java is a general programming language that includes some support for agents, namely to migrate code to clients as *applets* and dynamically bind to them at run-time. *Servlets* are similar to *applets*, but are designed to execute at the server without a user-interface.

Although both Java and Telescript are becoming popular as mobile object systems, none supports orthogonal persistence. For example, an *applet* is a normal Java class (with limited functionality for security reasons) so it cannot carry data with it except static variables. Telescript has some support for persistence, but it is not an orthogonal persistent programming language.

This is despite the fact that many agent applications require access to databases, carry data with them, or even more likely need both. (If mobile objects are not accessing, collecting and returning to their hosts with data, why do they need to travel between computers in the first place ?) Built-in support for persistence would help enormously when writing this kind of agent application.

The agent community is finally recognising this limitation. For example, the IBM Research Lab in Tokyo is currently working on *aglets* [IBM96b]. An *aglet* is a mobile object written in Java but carries its code as well as its state. The *aglets framework* also includes JoDax, a data access library to support the development of distributed database applications based on *aglets*.

The persistent community is particularly well positioned to contribute to this novel research area. HIPPO [Con96] is a new persistent programming system that will allow the sharing of code over the Internet. PJava [AJDS96, ADJ+96] is an orthogonally persistent version of Java that treats a class like any other object in the language; in particular, a class can be made persistent and copied between PJava stores. We have also given our first steps towards this goal by describing the main opportunities and challenges facing the integration between persistence and mobility [MdSA96a, MdS97, MdSRdS97, RdSMdSD97b, RdSMdSD97a].

8.2.2 Implementation Issues

The performance of Napier/RPC can be greatly improved. We separate the implementation issues proposed as future work into several experiments.

1. Optimize the current implementation of Napier/RPC by following a detailed pursuit of the sources of cost.
2. Re-implement the mechanisms proposed at the run-time system level below the type-checked boundary of Napier88.
3. Re-design and re-implement the mechanisms at the language level in a compiled (as opposed to interpreted) version of Napier88 or another persistent programming language.
4. Re-invent persistent spaces with a different architecture.

The first approach re-uses the current implementation. For example, in the initial RPC measured in section 7.2.1 the marshalling time is severe. By modifying the code to omit everything else, it should be possible to measure precisely how much is it and where these costs come from. One serious cost must be the access to each element of an object using a function to circumvent the type system; another to manipulate the data structures used to determine if an object has already been packed. These costs could be much reduced.

However, Napier88 is interpreted and its speed is 3 orders of magnitude slower than C (see table 7.4). Using this approach, Napier/RPC would never have a performance comparable with other RPC systems implemented in compiled languages.

The second approach — working at the run-time system level — avoids paying the price for type-safety and orthogonal persistence. Writing Napier/RPC in C leads to a much faster (roughly 400 times) and more efficient marshalling and unmarshalling (proportional to N log(N), not N^2). This would bring the 38 minutes for un-packing the Explorer index presented in table 7.8 to half a second or even less.

However, the second approach has disadvantages as well. It makes the job of implementing the RPC much harder and more unreliable because it loses the support of persistence and type-safety. It also specialises the RPC to one particular implementation of Napier88; if the language implementation later changes (likely in a research language) then the marshalling and unmarshalling routines have to be re-written.

A promising compromise retaining type-checked support for most of the code would be to define an *application programming interface* (API) to the Napier88 abstract machine available to other system implementors. This API would offer a service similar to the current version of the Napier88 special compiler that accepts low-level object manipulation, extended with stable object identifiers and eventually other basic services — such as configurable marshalling and unmarshalling executing at the speed of C. (This API could be investigated as part of any of the experiments proposed.)

The third approach is just taking advantage of a faster language implementation. From the RPC point of view it has few opportunities for research on novel distribution techniques and would be interesting only if the new implementation of the language brings also additional features.

The fourth approach is to experiment with novel store architectures that facilitate or even eliminate the need for marshalling and unmarshalling. For example, stores are first-class language values in the Feynman system [BP93]. This means that new stores can be created at run-time and become part of the existing store, forming an *hierarchy of object stores* [HP90].

Stores have the advantage of being marshalled structures already. A store could be sent as an argument to a remote procedure without requiring any marshalling at all [Bla95]. Instead, the marshalling cost is paid during normal program execution as happens already to read/write objects from/to the stable store.

8.2.3 Example Applications

The extensions made to the Library Explorer were an interesting experiment. However, a number of other distributed applications are needed to cover a wide spectrum of all persistent applications. Here we only describe *library installation*, an application area that is promising not only for testing the models proposed but also in its own right.

Library Installation

Software libraries for persistent stores, such as the Glasgow Libraries [WWP+95], are currently installed in each user's store by running programs. Libraries are typically large, for example, Glasgow Libraries needs 13 MB of store space and the Workshop [SWA+96] another 13 MB. Installation is slow, especially since the entire library has to be installed for every update, even though the user is likely to use only a small part of it during its lifetime.

It would be useful if users could install only those parts of each library they actually need. This is currently very difficult because the relationships between each sub-part in a library are very complex; the user prefers to install the entire library instead of understanding the dependencies between them.

The *Library Installer* is a proposed tool for Napier88 that would extend the existing Distributed Explorer. Using the Installer, procedures could be found and installed *on-demand* in the local store, without requiring any program to be explicitly executed.

Only the procedures installed and those required by these — that are not already present and are not substitutable — would be copied to the user's store. New versions would be propagated using a combination of substitution and persistent spaces. The crucial point is

that *the analysis of the call graph and the identification of which bindings to form would be entirely automated.*

A very first prototype of the Installer migrating one procedure was presented to the $FIDE_2$ Final Review Meeting that took place in Glasgow during September 1995. In that demonstration, an application programmer used the Explorer to look for a procedure offering some functionality. When the procedure was found, the programmer checked its documentation and decided to *download the value of the procedure itself to the local store.*

The procedure used in the demonstration contained a number of references to other objects in the store, including other procedures. Some of these objects were copied, but those that already existed in the user store migrated by substitution. A general, fully automated mechanism for library installation from a "Definitive Library Server" is a research topic. It would need to be aware of versions and integrated with the configuration and build tools [Sjø93, SWA+95].

8.2.4 Heterogeneity and Inter-operability

We now identify two compromises between what can be achieved by exploiting the features in one system and what can be achieved by mixing different systems.

Heterogeneity is the ability to implement Napier/RPC, substitution and persistent spaces in other persistent languages. For example, in Tycoon [MMM93, MMS94] or Persistent Java [AJDS96, ADJ+96]. In this context, the research work by Kato and Ohori with multi-language persistent type systems [KO92] is highly relevant.

Inter-operability is the ability of the mechanisms proposed to communicate with other (equivalent) mechanisms, eventually hosted by different programming languages. For example, with Sun/RPC [Sun93b], Tycoon/RPC [Mat96] or the novel distribution mechanisms proposed for Persistent Java [SA97, Spe97]. This would require to establish a sub-set of minimum functionality for both mechanisms, eventually by negotiation.

8.2.5 Further Speculation

Network bandwidth and CPU power will continue to increase dramatically as they have over the last few years. In contrast, network latency will always be limited by the speed of light. The *slowness of light* [Car96] can be acceptable within a local area network and for many Internet applications (such as the Web). However, it is clearly *not* appropriate for all world-wide distributed (global) applications and even less for "space applications" (e.g., satellites) [Kat96].

In such large-scale, geographically distributed environments, applications will have to be aware of locality. The number of round-trip communications will need to be reduced, presumably by achieving more per trip (such as caching and pre-fetching). The Inter-

net shows this trade-off very well: the most successful distributed applications are those that *exploit locality* like electronic mail and newsgroups. Even the Web uses physical machine addresses, fire-walls against intruders, caching mechanisms and ways of organizing information.

New models of global computation will have to be found and global programming systems will eventually be built to support the development of global applications. The concept of "site" and "domain" will be as important in these global systems as "object" and "class" are today for object-oriented ones. First steps have already been made, including those of Cardelli [Car96] and Connor [Con96]. Techniques like substitution and persistent spaces will become more appropriate because they were designed for environments in which latency dominates.

8.3 Goals Achieved

In this thesis we developed a number of extensions to the basic RPC mechanism. It is interesting to compare this experience with our initial list of IPC design issues presented in section 2.3. For each of these IPC issues we ask the following question: "Have we dealt with this IPC issue ?" If the answer is "yes", then we report how the models proposed solved it.

- *Understandability* (page 21) — Yes. The models proposed are all based on simple primitives that are easy to understand by typical application programmers. The example application presented in section 7.1 shows how these were used to extend an existing persistent application with support for distribution.

- *Type-safety* (page 22) — Yes. Type-safety is not only enforced between autonomous stores but it also does not require any extra programmer effort. The kind of type-safety we provide emphasizes autonomy because communicating stores only have to cross type-check once per *type session* and sessions are typically long-lived.

- *Type-completeness* (page 23) — Partly. Napier/RPC can pass almost all Napier88 data types, including procedures. However, for the purposes of this thesis we have excluded *abstract data types* (ADTs) and *threads*, implemented as an ADT in Napier88. The problem is that, while we base our type-checking on *structural equivalence*, ADTs are based on *name equivalence* and thus require a completely different approach. (This is part of future work; however, for reasons of space we have decided to leave it out from section 8.2.)

- *Synchronisation* (page 24) — Yes. Napier/RPC provides this functionality strictly. A more flexible synchronisation mechanism is provided by persistent spaces.

- *Efficiency, Performance and Scalability* (page 25) — Perhaps. Although the current Napier/RPC has problems with performance as a consequence of the Napier88 performance itself, there is nothing against an efficient implementation of the models

proposed in this thesis. We have described several ways of improving performance in section 8.2.2 but their actual effectiveness can only be confirmed by (re-)implementing the models proposed.

- *Replication and Caching* (page 26) — Yes. Persistent spaces were explicitly designed to provide a simple but flexible mechanism to maintain local copies of remote persistent objects, although not to maintain their strict consistency. Replication can be supported by using two spaces, one from a server to a client, another from the client back to the server.

- *Heterogeneity* (page 29) — No. Napier/RPC was designed and implemented for Napier88 with persistence in mind. The models proposed can be implemented in any other persistent language offering the same basic characteristics as Napier88. Even though the models are still suitable for traditional (non-persistent) programming languages, full advantage can only be taken in a persistent environment.

- *Fault-tolerance* (page 30) — Yes. All three models proposed focus on autonomy above all and thus provide a programmer interface and behaviour suitable for environments with partial failures. For example, a failure is reported by Napier/RPC 2.2 if the publisher store does not respond or if a type mismatch occurs; migration by substitution is based on long-lived type sessions with short set-up time; and persistent spaces are intended to be used for occasional connections between otherwise independent stores.

Furthermore, we have addressed two other goals that usually are not taken into consideration as IPC design issues.

- *Persistence* — Yes ! Persistence is not usually listed as an IPC design issue because databases are supposed to solve that problem separately. However, as persistence becomes more and more popular as part of the programming language, IPC mechanisms will have to take into consideration the new opportunities and challenges posed by persistence.

- *Higher-order* — Yes ! Modern programming languages such as Java [AG96] promote code to first-class status. As a result, the possibility to migrate code like any other data type will quickly become an issue for modern IPC mechanisms. Mobile agents (see section 8.2.1) will introduce further pressure.

8.3.1 Thesis Statement Revisited

Overall the thesis statement of proposing and implementing models for higher-order, type-safe, distributed computation over autonomous persistent stores that are realistic, understandable and general (see section 1.2) has been achieved.

- *Simple* — The models are easy to understand and use.

The source code examples presented throughout the thesis support this claim. In addition, our own exprience and the experience of others with examples of distributed persistent applications such as the Library Explorer further confirm the simplicity of these models.

- *General*—The models are not specialised to any specific use.

  Although we do make assumptions about the kind of application and its requirements, other IPC mechanisms concentrate on more limiting "horizontal" characteristics of a distributed application such as performance or heterogeneity. The models proposed are also not intended for certain data types like the Web [Wor96], OLE DB to access relational databases [Bla96b, Bla96a] or compound documents [App96].

- *Realistic*—The models were used to build real distributed persistent applications by typical programmers.

  The overhead imposed by distribution seems acceptable compared with the original application. Furthermore, there is nothing fundamental preventing a highly-efficient implementation of these models.

The research work described in this thesis will become ever more important. The issues listed as future work are clearly relevant, but others will no doubt arise. In any case, the research we have pursued will prove essential to cope with the new kind of world-wide persistent applications unimaginable even a few years ago.

Bibliography

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, November 1983.

[ABC90] R. Alonso, D. Barbara, and L. Cova. Using stashing to increase node autonomy in distributed file systems. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems (October 9-11, 1990, Huntsville, Alabama)*. IEEE Computer Society Press, 1990.

[ABC+93] M.P. Atkinson, P.J. Bailey, D. Christie, K. Cropper, and P. Philbrow. Towards bulk type libraries for Napier88. Technical Report FIDE/93/78, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1993.

[ABGO95] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An introduction to the database programming language Fibonacci. *VLDB Journal*, 4(3), 1995.

[ABM88] M.P. Atkinson, O.P. Buneman, and R. Morrison. Binding and type checking in database programming languages. *The Computer Journal*, 31(2):99–109, April 1988.

[ACC82] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-algol: An algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1982.

[ADG+89] A. Albano, A. Dearle, G. Ghelli, C. Marlin, R. Morrison, R. Orsini, and D. Stemple. A framework for comparing type systems for database programming languages. In R. Hull, R. Morrison, and D. Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages (Salishan Lodge, Gleneden Beach, Oregon, June 1989)*, pages 170–178. Morgan Kaufmann Publishers, 1989.

[ADJ+96] M.P. Atkinson, L. Daynès, M. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Record*, December 1996.

[AE90] M.P. Atkinson and A. England. Towards new architectures for distributed autonomous database applications. In Rosenberg and Keedy [RK90], pages 356–377.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison Wesley, 1996. ISBN 0-201-63455-4.

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (San Jose, California, May 22-25, 1995)*, SIGMOD Record, pages 23–34, June 1995.

[AJDS96] M.P. Atkinson, M. Jordan, L. Daynès, and S. Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system. In Atkinson et al. [AMB96].

[AKP+94] M.P. Atkinson, G. Kirby, P. Philbrow, J. W. Schmidt, C.A. Waite, R.C. Welland, and I. Wetzel. Deliverable FD5: Persistent workbench research. Technical report, ESPRIT Basic Research Action, Project Number 6309 — $FIDE_2$, August 1994.

[AM85] M.P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 4(7):539–559, October 1985.

[AM86] M.P. Atkinson and R. Morrison. Integrated persistent programming systems. In B.D. Shriver, editor, *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, IIA Software Track (7th–10th January 1986)*, pages 842–854, 1986.

[AM88] M.P. Atkinson and R. Morrison. Types, bindings and parameters in a persistent environment. In M.P. Atkinson, O.P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, Topics in Information Systems, series editors M.L. Brodie, J. Mylopoulos and Schmidt, J.W., chapter 1, pages 3–20. Springer-Verlag, 1988. Edited Papers from the Proceedings of the First Workshop on Persistent Object Systems (Appin, Scotland, August 1985).

[AM90] M.P. Atkinson and R. Morrison. Polymorphic names and iterations. In F. Bancilhon and O.P. Buneman, editors, *Advances in Database Programming Languages*, ACM Press, Frontier Series, pages 241–256. Addison-Wesley Publishing Company and ACM Press, 1990. Edited Proceedings of the Workshop on Database Programming Languages (Roscoff, Brittany, France, September 1987).

[AM95] M.P. Atkinson and R. Morrison. Orthogonal persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[AMB96] M.P. Atkinson, D. Maier, and V. Benzaken, editors. *Proceedings of the Seventh International Workshop on Persistent Object Systems (Cape May, New Jersey, USA, May 29-31, 1996)*. Morgan Kaufmann Publishers, 1996.

[AMP87] M.P. Atkinson, R. Morrison, and G.D. Pratten. PISA: A persistent information space architecture. *ICL Technical Journal*, 5(3):477–491, May 1987.

[APM89] APM — Architecture Projects Management Limited. *The ANSA Reference Manual, Release 01.01*, July 1989.

[App96] Apple Computer, Inc. *Welcome to OpenDoc Web!*, 1996. http://www.-opendoc.apple.com/.

[AT96] Paolo Atzeni and Val Tannen, editors. *Proceedings of the Fifth International Workshop on Database Programming Languages (Gubbio, Umbria, Italy, 6th-8th September 1995)*, Electronic Workshops in Computing. Springer-Verlag, 1996.

[ATK92a] A.L. Ananda, B.H. Tay, and E.K. Koh. A survey of asynchronous remote procedure calls. *Operating Systems Review*, 26(2):92–109, April 1992.

[Atk92b] M.P. Atkinson. Persistent foundations for scalable multi-paradigmal systems. In Özsu et al. [ÖDV92]. Invited paper.

[BALL89] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *Operating Systems Review*, 23(5):102–113, December 1989.

[BC95a] K. Bharat and L. Cardelli. Distributed applications in a multimedia setting. In *Proceedings of the First International Workshop on Hypermedia Design (Montpelier, France, 1995)*, pages 185–192, 1995.

[BC95b] K. Bharat and L. Cardelli. Migratory applications. In *Proceedings of ACM Symposium on User Interface Software and Technology '95 (Pittsburgh, PA, Nov 1995)*, pages 133–142, 1995.

[BCL+87] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo, and M. Schwartz. A remote procedure call facility for interconnecting heterogeneous operating systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[Ben87] J.K. Bennett. The design and implementation of distributed smalltalk. In *Proceedings of the Second Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (Orlando, Florida, Oct. 1987)*, 1987.

[Ben90] J.K. Bennett. Experience with Distributed Smalltalk. *Software Practice and Experience*, 20(2):157–180, February 1990.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[BHL95] B. Blakeley, H. Harris, and J. R. T. Lewis. *Messaging and Queuing Using the MQI: Concepts and Analysis, Design and Development.* McGraw-Hill, 1995. ISBN 0-07-005730-3.

[Bir88] A. Birrell. Position paper. In *Proceedings of the 1988 ACM SIGOPS European Workshop*, 1988.

[Bir93a] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 103, December 1993.

[Bir93b] K. Birman. A response to Cheriton and Skeen's criticism of casual and totally ordered communication. Technical Report Cornell TR 93-1390, Dept. of Computer Science, Cornell University, October 1993.

[BJW87] A. Birrell, M. Jones, and E. Wobber. A simple and efficient implementation for small databases. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987.

[Bla95] D. Blakeman. Private communication, 1995.

[Bla96a] José A. Blakeley. Data access for the masses through OLE DB. In Jagadish and Mumick [JM96], pages 161–172.

[Bla96b] José A. Blakeley. OLE DB: A component DBMS architecture. In Su [Su96], pages 203–204.

[BLL+88] A. Black, E. Lazowska, H. Levy, D. Notkin, J. Sanislo, and J. Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, March 1988.

[BN84] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BNOW93] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, December 1993.

[BP93] D. Blakeman and M. Powell. The Feynman persistent application support environment. Technical report, Software Engineering Tools Group, Department of Computation, UMIST, 1993.

[BR90] A.L. Brown and J. Rosenberg. Persistent object stores: An implementation technique. In Dearle et al. [DSZ90].

[Bro88] A.L. Brown. *Persistent Object Stores.* PhD thesis, University of St Andrews, 1988.

[Bro93] J.C. Brown. A library explorer for the Napier88 Glasgow Libraries. Master's thesis, Department of Computing Science, University of Glasgow, September 1993.

[BTV96] J. Baumann, C. Tschudin, and J. Vitek, editors. *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems (Linz, Austria, July 8-9, 1996)*. dpunkt, 1996.

[Bul96] BulletProof Corporation. *MarketPage from BulletProof*, 1996. http://www.bulletproof.com/Marketpage/.

[CAL+94] R. Cooper, M.P. Atkinson, D. Lavery, M. Mira da Silva, G. Montgomery, P. Philbrow, A. Pirmohamed, T. Printezis, A. Serrano, C.A. Waite, and R.C. Welland. *The Glasgow Libraries Reference Manual Version 1.1.* Department of Computing Science, University of Glasgow, December 1994. Compatible with Napier88 Release 2.0.

[Car95a] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995. A preliminary version appeared in Proceedings of the 22nd ACM Symposium on Principles of Programming Languages.

[Car95b] L. Cardelli. Mobile computation. In *ARPA/NSF Workshop on Foundational Studies for Software Engineering (Stanford, September, 1995)*, 1995. Position paper.

[Car96] L. Cardelli. *Global Computation.* Digital Equipment Corporation, Systems Research Center, 1996.

[CBC+90a] R.C.H. Connor, A.L. Brown, R. Carrick, A. Dearle, and R. Morrison. The persistent abstract machine. In J. Rosenberg and D.M. Koch, editors, *Persistent Object Systems*, pages 353–366. Springer-Verlag, 1990.

[CBC+90b] R.C.H. Connor, A.L. Brown, Q.I. Cutts, A. Dearle, R. Morrison, and J. Rosenberg. Type equivalence checking in persistent object systems. In Dearle et al. [DSZ90], pages 154–167.

[CBM96] R.C.H. Connor, D. Balasubramaniam, and R. Morrison. Investigating extension polymorphism. In Atzeni and Tannen [AT96].

[CC91] R.S. Chin and S.T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), March 1991.

[CCM95] Q.I. Cutts, R.C.H. Connor, and R. Morrison. The PamCase machine. In M.P. Atkinson, editor, *Fully Integrated Data Environments*, chapter 2.1.3. Springer-Verlag, 1995.

[CDK94] G. Colouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, second edition, 1994.

[CDKM89] Q.I. Cutts, A. Dearle, G.N.C. Kirby, and C.D. Marlin. WIN: A persistent window management system. Technical Report PPRR-73-89, Universities of Glasgow and St Andrews, 1989.

[CDMB90] R.C.H. Connor, A. Dearle, R. Morrison, and A.L. Brown. Existentially quantified types as a database viewing mechanism. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Proceedings of the Second International Conference on Extending Database Technology (Venice, Italy, March 1990)*, number 416 in Lecture Notes in Computer Science, pages 301–315. Springer-Verlag, 1990.

[Che93] W.K. Cheng. Distributed object database management systems. *Journal of Object-Oriented Programming*, March-April 1993.

[CKW96] Oliver Ciupke, Dietmar Kottmann, and Hans-Dirk Walter. Object migration in non-monolithic distributed applications. In ICDCS-96 [ICD96].

[CMM91] R.C.H. Connor, D. McNally, and R. Morrison. Subtyping and assignment in database programming languages. In Kanellakis and Schmidt [KS91], pages 305–324. Proceedings of the Third International Workshop on Database Programming Languages (Nafplion, Greece, 27th–30th August 1991).

[CO94] S.C. Crawley and M.J. Oudshoorn. Orthogonal persistence and Ada. In *Proceedings TRI-Ada'94 Nov 6-11 1994*, pages 298–308. Association for Computing Machinery, 1994.

[Coh96] Norman H Cohen. *Ada as a second language.* McGraw-Hill, 1996.

[Con91] R.C.H. Connor. *Types and Polymorphism in Persistent Programming Systems.* PhD thesis, University of St Andrews, 1991.

[Con96] R.C.H. Connor. *Welcome to the HIPPO home page.* University of St Andrews, 1996. http://grappa.dcs.st-and.ac.uk/HIPPO/.

[COO96] *Proceedings of the 2nd Conference on Object-Oriented Technologies and Systems (June 17-21, 1996, Toronto, Ontario, Canada)*, 1996.

[Cra93] D.H. Craft. A study of pickling. *Journal of Object-Oriented Programming*, January 1993.

[Cut93] Q.I. Cutts. *Delivering the Benefits of Persistence to System Construction and Execution.* PhD thesis, University of St Andrews, 1993.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[Dan95] J. Daniels. Position paper. In *Proceedings of the OOPSLA'95 Workshop on Building Large Distributed Software Systems Using Objects*, 1995.

[DC93] J. Daniels and S. Cook. Strategies for sharing objects in distributed systems. *Journal of Object-Oriented Programming*, January 1993.

[DdBF+94] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, and J. Rosenberg. Grasshopper: An orthogonally persistent operating system. *Computer Systems*, 7(3):289–312, 1994.

[Dea89] A. Dearle. Environments: A flexible binding mechanism to support system evolution. In B.H. Shriver, editor, *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, Volume II Software Track (January 1989)*, pages 46–45, 1989.

[Deu91] O. Deux. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[DG92] L. Daynès and O. Gruber. Nested actions in Eos. In A. Albano and R. Morrison, editors, *Proceedings of the Fifth International Workshop on Persistent Object Systems (San Miniato, Italy, 1st–4th September 1992)*, Workshops in Computing, pages 144–163. Springer-Verlag in collaboration with the British Computer Society, 1992.

[DH93] J.P. Deschrevel and A.J. Herbert. *The ANSA Model of Federation and Trading.* Architecture Projects Management Ltd., Cambridge (UK), February 1993.

[Dij68] E.W. Dijkstra. Cooperating sequential processes. In Genyus, editor, *Programming Languages.* Academic Press, New York, 1968.

[DPS+94] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (Santa Cruz, California, December 1994)*, pages 2–7. IEEE Computer Society Press, 1994.

[DRV91] A. Dearle, J. Rosenberg, and F. Vaughan. A remote execution mechanism for distributed homogeneous stable stores. In Kanellakis and Schmidt [KS91]. Proceedings of the Third International Workshop on Database Programming Languages (Nafplion, Greece, 27th–30th August 1991).

[DSZ90] A. Dearle, G.M. Shaw, and S.B. Zdonik, editors. *Proceedings of the Fourth International Workshop on Persistent Object Systems, Their Design, Implementation and Use (Martha's Vineyard, USA, September 1990).* Morgan Kaufmann Publishers, 1990.

[ESS91] J.L. Eppinger, N. Saxena, and A.Z. Spector. *Transactional RPC.* Transarc Corporation, 1991.

[Eva96] Huw Evans. Private communication, 1996.

[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 (Manhattan, New York City, USA, 30th August–1st September 1993).* Springer-Verlag in collaboration with the British Computer Society, 1993.

[FD94] A. Farkas and A. Dearle. The Octopus model and its implementation. *Australian Computer Science Communications*, 16(1), 1994.

[Fre96] E. Freeman. *Linda Group.* Department of Computer Science, University of Yale, 1996. http://www.cs.yale.edu/HTML/YALE/CS/Linda/-linda.html.

[Gib87] P. B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, 13(1):77–87, January 1987.

[GNSP94] Y-S. Gutfreund, J. Nicol, R. Sasnett, and V. Phuah. WWWinda: An orchestration service for WWW browsers and accessories. In *Proceedings of the Second International WWW Conference: Mosaic and the Web*, 1994.

[Gru92] O. Gruber. *Eos, an Environment for Persistent and Distributed Applications over a Shared Object Space.* PhD thesis, Université Pierre et Marie Curie, Paris VI, France, December 1992.

[Ham84] K.G. Hamilton. *A Remote Procedure Call System.* PhD thesis, University of Cambridge Computer Laboratory, 1984.

[Han72] P. Brinch Hansen. Distributed processes, a concurrent programming concept. *Communications of the ACM*, 21(11):934–941, 1972.

[Han73] P. Brinch Hansen. *Operating System Principles.* Prentice Hall, 1973.

[Han75] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, 1975.

[Har92] Samuel P. Harbison. *Modula-3.* Prentice Hall, 1992.

[HKM+88] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[HL82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[HP90] C. Harrison and M. Powell. A modular persistent store. In Dearle et al. [DSZ90].

[HPM93] G. Hamilton, M.L. Powell, and J.G. Mitchell. Subcontract: A flexible base for distributed programming. Technical Report SMLI TR-93-13, Sun Microsystems Laboratories, 1993.

[IBM94] IBM. *MQSeries: Distributed Queue Management Guide*, third edition, June 1994.

[IBM95] IBM. *MQSeries: Application Programming Guide*, third edition, February 1995.

[IBM96a] IBM Corporation. *IBM MQSeries product family home page: Commercial Messaging Software*, 1996. http://www.hursley.ibm.com/mqseries/.

[IBM96b] IBM Tokyo Research Lab. *Aglets Workbench: Programming Mobile Agents in Java*, 1996. http://www.trl.ibm.co.jp/aglets/.

[ICD96] *Proceedings of the 16th International Conference on Distributed Computing Systems (Hong-Kong, May, 1996)*. IEEE Computer Society Press, 1996.

[JLHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[JM96] H.V. Jagadish and I.S. Mumick, editors. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (Montreal, Quebec, Canada, June 4-6, 1996)*, SIGMOD Record, June 1996.

[JSS94] B. Janssen, D. Severson, and M. Spreitzer. *ILU 1.6.4 Reference Manual.* Xerox Corporation, May 1994.

[JV95] M. Jordan and M. Van De Vanter. Software configuration management in an object-oriented database. In *Proceedings USENIX 1995 Conference on Object-Oriented Technologies (Monterey, California, June 26-29, 1995)*, 1995.

[Kap95] Frank Kappe. A scalable architecture for maintaining referential integrity in distributed information systems. *The Journal of Universal Computer Science*, 1(2), February 1995.

[Kat96] K. Kato. Private communication, 1996.

[KBC+94] G.N.C. Kirby, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, V.S. Moore, R. Morrison, and D.S. Munro. The Napier88 standard library reference manual version 2.2. Technical Report FIDE/94/105, ESPRIT Basic Research Action, Project Number 6309—$FIDE_2$, 1994.

[Kir93] G.N.C. Kirby. *Reflection and Hyper-Programming in Persistent Programming Systems.* PhD thesis, University of St Andrews, 1993.

[KKM94] K. Kono, K. Kato, and T. Masuda. Smart remote procedure calls: Transparent treatment of remote pointers. In *Proceedings of the 14th International Conference on Distributed Computing Systems (Poznan, Poland, June 21–24, 1994)*. IEEE Computer Society Press, 1994.

[KMK96] K. Kono, T. Masuda, and K. Kato. An implementation method of migratable distributed objects using an RPC technique integrated with virtual memory management. In P. Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP) (Linz, Austria, July 10-12, 1996)*, Lecture Notes in Computer Science, pages 295–315. Springer-Verlag, 1996.

[Kna94] F. Knabe. Function transmission for a distributed higher-order language. Extended Abstract, November 1994.

[Kna95] F. Knabe. *Language Support for Mobile Agents.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, USA, December 1995.

[KO92] K. Kato and A. Ohori. An approach to multilanguage persistent type system. In R. Morrison and M.P. Atkinson (minitrack coordinators), editors, *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, Volume II, Software Technology, Persistent Object Systems*, pages 810–819, 1992.

[KOMM93] K. Kato, A. Ohori, T. Murakami, and T. Masuda. Distributed C language based on a higher-order RPC technique. *JSSST*, 5:119–143, 1993.

[KS91] P. Kanellakis and J.W. Schmidt, editors. *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, 1991. Proceedings of the Third International Workshop on Database Programming Languages (Nafplion, Greece, 27th–30th August 1991).

[KSD+90] B. Koch, T. Schunke, A. Dearle, F. Vaughan, C. Marlin, R. Fazakerley, and C. Barter. Cache coherency and storage management in a persistent object system. In Dearle et al. [DSZ90].

[KTM+96] K. Kato, K. Toumura, K. Matsubara, S. Aikawa, J. Yoshida, K. Kono, K. Taura, and T. Sekiguchi. Protected and secure mobile object computing in PLANET. In Baumann et al. [BTV96].

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In Jagadish and Mumick [JM96].

[Lar96] P. Larsson. TkWin: A modern GUI for Napier88. Master's thesis, Department of Computing Science, University of Gotebörg, Sweden, 1996. Performed at Department of Computing Science, University of Glasgow.

[Lav95a] D. Lavery. The design of effective software visualizations for persistent programming languages. Technical Report FIDE/95/116, ESPRIT Basic Research Action, Project Number 6309—$FIDE_2$, 1995.

[Lav95b] D. Lavery. Towards visualizing persistent stores. Technical Report FIDE/95/122, ESPRIT Basic Research Action, Project Number 6309—$FIDE_2$, 1995.

[LBG+88] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. E. Weihl. Communication in the Mercury system. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988.

[LDS92] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In Özsu et al. [ÖDV92].

[Lie93] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188. Association for Computing Machinery, 1993.

[Lis84] B. Liskov. Overview of the Argus language and system. MIT Programming Methodology Group Memo 40, MIT, February 1984.

[Lis88] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. ObjectStore. *Communications of the ACM*, 34(10):51–63, October 1991.

[Loo93] M. Loomis. Distributed object databases. *Journal of Object-Oriented Programming*, March-April 1993.

[Lop95] C. Lopes. Graph-based optimizations for parameter passing in remote invocations. In *Proceeding of the 4th International Workshop on Object Orientation in Operating Systems (Lund, Sweden, August 1995)*. IEEE Computer Society Press, 1995.

[Lop96] C. Lopes. Adaptive parameter passing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (Kanazawa, Japan, Mar. 11-15, 1996)*, volume 1049 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Lot96] Lotus. *Notes: The Best in Messaging, Groupware, Internet Products and More*, 1996. http://www.lotus.com/allnotes/.

[LS83] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Lnaguage Design and Implementation*, pages 260–267, 1988.

[Mar96] Marimba. *Castanet*, 1996. http://www.marimba.com/products/castanet.-html.

[Mat96] Bernd Mathiske. *Mobility in Persistent Object Systems*. PhD thesis, Computer Science Department, Hamburg University, Germany, May 1996. In German.

[MBC+90] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby, J. Rosenberg, and D. Stemple. Protection in persistent object systems. In Rosenberg and Keedy [RK90], pages 48–66.

[MBC+94] R. Morrison, A.L. Brown, R.C.H. Connor, Q.I. Cutts, A. Dearle, G.N.C. Kirby, and D.S. Munro. The Napier88 reference manual release 2.0. Technical Report FIDE/94/104, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994.

[MBCD89] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. The Napier88 reference manual. Technical Report PPRR-77-89, Universities of Glasgow and St Andrews, 1989.

[MdS95a] M. Mira da Silva. Automating type-safe RPC. In O.A. Bukhres, M.T. Özsu, and M.C. Shan, editors, *Proceedings of The Fifth International Workshop on Research Issues on Data Engineering: Distributed Object Management (Taipei, Taiwan, 6th–7th March 1995)*, pages 100–107. IEEE Computer Society Press, 1995.

[MdS95b] M. Mira da Silva. Programmer's manual to Napier88/RPC 2.2. Technical Report FIDE/95/133, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1995.

[MdS97] M. Mira da Silva. *Mobile Object Systems*, chapter Mobility and Persistence. Number 1222 in Lecture Notes in Computer Science. Springer-Verlag, 1997.

[MdSA96a] M. Mira da Silva and M. Atkinson. Combining mobile agents with persistent systems: Opportunities and challenges. In Baumann et al. [BTV96].

[MdSA96b] M. Mira da Silva and M.P. Atkinson. Higher-order distributed computation over autonomous persistent stores. In Atkinson et al. [AMB96].

[MdSAB96] M. Mira da Silva, M.P. Atkinson, and A. Black. Semantics for parameter passing in a type-complete persistent RPC. In ICDCS-96 [ICD96].

[MdSRdS97] M. Mira da Silva and A. Rodrigues da Silva. Insisting on persistent mobile agent systems. In R. Popescu-Zeletin and K. Rothermel, editors, *Proceedings of the First International Workshop on Mobile Agents (Berlin, Germany, April 7-8, 1997)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[Mel96] Jim Melton. SQL3 update. In Su [Su96], pages 666–672.

[MMM93] B. Mathiske, F. Matthes, and S. Mussig. The Tycoon system and library manual. Technical Report DBIS Tycoon Report 212-93, Computer Science Department, University of Hamburg, December 1993.

[MMS94] F. Matthes, S. Müßig, and J.W. Schmidt. Persistent polymorphic programming in Tycoon: An introduction. Technical Report FIDE/94/106, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1994.

[MMS95] B. Mathiske, F. Matthes, and J.W. Schmidt. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems (Naharia, Israel, June 1995)*, 1995.

[MMS96] B. Mathiske, F. Matthes, and J.W. Schmidt. Scaling database languages to higher-order distributed programming. In Atzeni and Tannen [AT96].

[MP85] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 37–51, New Orleans, January 1985.

[MT86] S.J. Mullender and A.S. Tanenbaum. The design of a capability-based distributed operating system. *The Computer Journal*, 29(4):289–300, 1986.

[Mul93] S.J. Mullender, editor. *Distributed Systems.* ACM Press, 2nd edition, 1993.

[Mun93] D.S. Munro. *On the Integration of Concurrency, Distribution and Persistence.* PhD thesis, University of St Andrews, 1993.

[Nel81] B. Nelson. *Remote Procedure Call.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1981.

[Net96] Netscape Communications Corporation. *Netscape Home Page*, 1996. http://www.netscape.com/.

[ODI96] ODI—Object Design, Inc. *ObjectStore for Smalltalk*, March 1996.

[ÖDV92] M.T. Özsu, U. Dayal, and P. Valduriez, editors. *Proceedings of the International Workshop on Distributed Object Management (Edmonton, Canada, 18th–21st August 1992)*, 1992.

[OK93] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Conference Record of the Twentieth Symposium on Principles of Programming Languages*, pages 99–112. Association for Computing Machinery, 1993.

[OMG95] OMG—Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA)*, 1995.

[OSF91] OSF—Open Software Foundation. *Remote Procedure Call in a Distributed Computing Environment: A White Paper*, 1991.

[Phi96] P. Philbrow. Private communication, 1996.

[Pre96] Premenos. *EDI Standards*, 1996. http://www.premenos.com/standards/.

[PS95] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management, (Kinross, UK, September 1995)*, 1995.

[RdSMdSD97a] A. Rodrigues da Silva, M. Mira da Silva, and José Delgado. AgentSpace: A framework for developing agent programming systems. In *Proceedings of the 4th International Conference on Intelligence in Services and Networks (Cernobbio, Como, Italy, May 27-29, 1997)*, 1997.

[RdSMdSD97b] A. Rodrigues da Silva, M. Mira da Silva, and José Delgado. Conceptual frameworks for Web information systems development. In *Proceedings of the 7th MINI EURO Conference (Bruges, March 24-27, 1997)*, 1997.

[RK90] J. Rosenberg and J.L. Keedy, editors. *Security and Persistence. Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information (Bremen, West Germany, 8–11 May 1990)*, Workshops in Computing. Springer-Verlag in collaboration with the British Computer Society, 1990.

[Ros90] John Rosenberg. The MONADS architecture: A layered view. In Dearle et al. [DSZ90], pages 215–225.

[Ros96] Davide Rossi. *Jada: multiple tuple spaces for Java a la Linda.* Department of Computer Science, University of Bologna, 1996. http://www.cs.unibo.-it/~rossi/jada/.

[RS87] L.A. Rowe and M. Stonebraker. The Postgres data model. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases (Brighton, England, 1987)*, pages 83–96, 1987.

[RWW96] R. Riggs, J. Waldo, and A. Wollrath. Pickling state in the Java system. In COOTS-96 [COO96].

[SA97] S. Spence and M. Atkinson. A scalable model of distribution promoting autonomy of and cooperation between PJava object stores. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences (Hawaii, USA, January 1997)*, 1997. To be published.

[SB89] Michael D. Schroeder and Michael Burrows. Performance of firefly RPC. *Operating Systems Review*, 23(5):83–90, December 1989.

[SBG+91] R.E. Strom, D.F. Bacon, A.P. Goldberg, A. Lowry, D.M. Yellin, and S.A. Yemini. *HERMES: A Language for Distributed Computing.* Series in Innovative Technology. Prentice Hall, 1991.

[SBH96] M. Strasser, J. Baumann, and F. Hohl. MOLE: A Java based mobile agent system. In Baumann et al. [BTV96].

[Sch77] J.W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, September 1977.

[SCWA94] D.I.K. Sjøberg, Q. Cutts, R. Welland, and M. Atkinson. Analysing persistent language applications. In M.P. Atkinson, V. Benzaken, and D. Maier, editors, *Proceedings of the Sixth International Workshop on Persistent Object Systems.* Springer-Verlag in collaboration with the British Computer Society, 1994.

[SG90] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 4(12):537–565, October 1990.

[Sjø93] D.I.K. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems.* PhD thesis, University of Glasgow, July 1993.

[Sjø96] D.I.K. Sjøberg. Private communication, 1996.

[SM86] R.J. Souza and S.P. Miller. Unix and remote procedure calls: A peaceful coexistence ? In *Proc. of the 6th Int. Conf. on Distributed Computing Systems (Cambridge, Massachusetts, May 19-23, 1986)*, pages 268–277, 1986.

[Spe82] A. Spector. Performing remote operations efficiently on a local computer network. *Communications of the ACM*, 25(4):246–259, April 1982.

[Spe97] S. Spence. Distribution strategies for Persistent Java. In M.P. Atkinson and M. Jordan, editors, *Proceedings of the First International Workshop on Persistence and Java (Drymen, Scotland, September 1996)*, 1997. To be published as a Sun Technical Report.

[SR86] M. Stonebraker and L.A. Rowe. The design of Postgres. In *Proceedings of the ACM SIGMOD 1986 Conference on the Management of Data (Washington, DC, May)*, pages 340–355, 1986.

[Sta82] J. Staunstrup. Message passing communication versus procedure call communication. *Software Practice and Experience*, 12:223–234, 1982.

[Su96] Stanley Y. W. Su, editor. *Proceedings of the Twelfth International Conference on Data Engineering (February 26 - March 1, 1996, New Orleans, Louisiana).* IEEE Computer Society Press, 1996.

[Sun93a] Sun Microsystems. *External Data Representation Standard: Protocol Specification*, 1993.

[Sun93b] Sun Microsystems. *RPC Programming Guide*, 1993.

[Sun93c] Sun Microsystems. *Transport Level Interface Programming*, 1993.

[Sun96a] Sun Microsystems. *Remote Method Invocation*, 1996. http://chatsubo.javasoft.com/current/rmi/.

[Sun96b] Sun Microsystems Inc. *Java IDL*, 1996. http://splash.javasoft.com/JavaIDL/pages/.

[Sun96c] Sun Microsystems Inc. *JDBC: A Java SQL API*, 1996. http://splash.javasoft.com/jdbc/.

[Sun96d] Sun Microsystems, Inc. *JEEVES: Java-powered Internet server and framework*, 1996. http://www.javasoft.com/products/jeeves/.

[SWA+95] D.I.K. Sjøberg, R.C. Welland, M.P. Atkinson, P. Philbrow, and C.A. Waite. Exploiting persistence in build management. Technical Report 6, University of Oslo, Department of Informatics, 1995.

[SWA+96] D.I.K. Sjøberg, R.C. Welland, M.P. Atkinson, P. Philbrow, C.A. Waite, and S.D. Macneill. The persistent workshop: A programming environment for Napier88. In *Proceedings of the 7th Nordic Workshop on Programming Environment Research (Aalborg, Denmark, 29-31 May, 1996)*, 1996.

[TA90] B.H. Tay and A.L. Ananda. A survey of remote procedure calls. *ACM Operating Systems Review*, 24(3), July 1990.

[TD94] I. Toyn and A. J. Dix. Efficient binary transfer of pointer structures. *Software Practice and Experience*, 24(11):1001–1023, 1994.

[THM+96] P. Trinder, K. Hammond, J. Mattson, A. Partridge, and S. Peyton Jones. GUM: A portable parallel implementation of Haskell. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation (Philadelphia, USA, 1996)*, 1996.

[Tra93a] Transarc Corporation. *Encina for the Solaris Operating Environment: Product Overview*, 1993.

[Tra93b] Transarc Corporation. *Encina for the Solaris Operating Environment: Recoverable Queuing Service*, 1993.

[Tri95] P. Trinder. Private communication, 1995.

[Wai88] F. Wai. *Distributed Concurrent Persistent Programming Languages: An Experimental Design and Implementation.* PhD thesis, University of Glasgow, April 1988.

[Wal96] J. Waldo. Distributed computing and persistence. JavaOne: Sun's Worldwide Java Developer Conference, 1996. Collection of slides.

[WB87] S. Wilbur and B. Bacarisse. Building distributed systems with remote procedure call. *Software Engineering Journal*, September 1987.

[WFN90] E.F. Walker, R. Floyd, and P. Neves. Asynchronous remote operation execution in distributed systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 253–259. IEEE, 1990.

[Whi94a] J.E. White. *Telescript Tecnhology: Scenes from the Electronic Marketplace.* General Magic, 1994.

[Whi94b] J.E. White. *Telescript Tecnhology: The Foundation for the Electronic Marketplace.* General Magic, 1994.

[Wik94] C. Wikström. Distributed programming in Erlang. In *Proceedings of the First International Symposium on Parallel Computation (PASCO'94)*, 1994.

[Wor96] World Wide Web Consortium. *HyperText Markup Language (HTML)*, 1996. http://www.w3.org/pub/WWW/MarkUp/MarkUp.html.

[WPA+95] C.A. Waite, P. Philbrow, M.P. Atkinson, R.C. Welland, D. Lavery, S.D. Macneill, T. Printezis, and R.C. Cooper. Programmer's Persistent Workshop: Principles and User Guide. Technical Report FIDE/95/125, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1995.

[WRW96] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In COOTS-96 [COO96].

[WWP+95] C.A. Waite, R.C. Welland, T. Printezis, A. Pirmohamed, P. Philbrow, G. Montgomery, M. Mira da Silva, S.D. Macneill, D. Lavery, C. Hertzig, A. Froggatt, R.L. Cooper, and M.P. Atkinson. Glasgow libraries for orthogonally persistent systems: Philosophy, organisation and contents. Technical Report FIDE/95/132, ESPRIT Basic Research Action, Project Number 6309—FIDE$_2$, 1995.

[WWWK94] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboratories, 1994.

Index