Exploiting Shared Memory for Protected Services

René W. Schmidt

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 94-06-03

This work was supported in part by the National Science Foundation (Grants No. CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, and Intel Corporation.

Exploiting Shared Memory for Protected Services

by

René Wenzel Schmidt

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

University of Washington

1994

Approved by____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree____

Date____

University of Washington

Abstract

Exploiting Shared Memory for Protected Services

by René Wenzel Schmidt

Chairperson of the Supervisory Committee: Professor Henry M. Levy
Department of Computer Science

The client/server model is commonly used to provide sharing and protection of data. Using this model, distrusted applications (clients) can access and possibly modify shared data, while guaranteeing data integrity. To ensure this data integrity, the shared data is managed by a server that is located in a private protection domain. Calls to the server domain utilize a protected procedure call mechanism, i.e., the calls execute within the server's domain. While this model provides safety, it can suffer in performance, due to the domain-crossing costs required by the protected procedure calls.

Porc is a toolkit for building object-based client/server applications. Its main goals are (1) to provide efficient access to protected objects and (2) to hide the protection boundary from clients. In Porc protected server objects are represented as local *proxy* objects in the client; the difference between a local object and a protected object is thus transparent to the client. This thesis describes three extensions of the Porc toolkit in order to provide fast access to protected objects.

The extensions are all based on read-only sharing. First, read-only methods must be defined, which are server procedures that do not modify server state. Read-only methods can be directly invoked by clients without requiring a protected procedure call. Second, version-based synchronization is needed; a mechanism, which can be used to efficiently synchronize read-only methods. Third, shared proxies are used to avoid the cost of local proxy creation in clients, making it faster to traverse complex pointer structures. Shared proxies also increase transparency by reducing pointer aliasing problems.

Table of Contents

List of Figures iii

List of Tables iv

Chapter 1: Introduction 1
1.1 Object-based RPC 2
1.2 Goals of the Toolkit 3
1.3 Exploiting Shared Memory 4
1.4 Related Work 6
1.5 Organization of the Thesis 7

Chapter 2: Object-based RPC 8
2.1 The Proxy Model 9
2.2 Capabilities 11
2.3 Portals and Channels 12
2.4 Summary 14

Chapter 3: Shared Proxies 15
3.1 Transparency 17
3.2 Channel-less Proxies 19
3.3 Binding 20
3.4 Implementation 21
3.5 Summary 23

Chapter 4: Read-only Methods 25
4.1 Parameter Passing and Validation 27
4.2 Implementation 29
4.3 Summary 30

Chapter 5: Version-based Synchronization 31
5.1 Using Locks 32
5.2 Version Numbering 34
5.3 Memory Management 36
5.4 Summary 38

Chapter 6: Evaluation 39
6.1 The Name Server 39
6.2 Experiments 42
6.3 Results 43

Chapter 7: Conclusion 46
7.1 Future Work 47

Bibliography 49

Appendix A: Version-based Synchronization: An Example 52
A.1 Memory Management 53
A.2 Read-only Methods 55
A.3 Write Methods 56

List of Figures

2.1 System Overview 10
2.2 Contents of a Capability 12

3.1 An object-based file server 16
3.2 Example of a shared data structure 18
3.3 Shared proxies, guards and protected objects 22
3.4 Stub code for a shared proxy 23

4.1 A client with read-only access to a server 26
4.2 Sample proxy code for a read-only method 29

5.1 A simple concurrent C++ class using the check-in/check-out scheme . . . 33
5.2 A simple concurrent C++ class using version numbering 35
5.3 Interface for the LockFreeObject class 36
5.4 Interface for the LockFreeMemPool class 38

6.1 The internal organization of the name server 40
6.2 Interface for the DirNode Class 41
6.3 Interface for the DirEntry Class 41

A.1 The List and ListElem Classes 53
A.2 The LockFreeObject Class 54
A.3 The LockFreeMemPool class 55
A.4 Implementation of the read-only methods 57
A.5 Implementation of getKey with incarnation check 58
A.6 Implementation of the write-methods 60

List of Tables

6.1 Performance Measurements 44

Acknowledgments

The author wishes to express sincere appreciation to Professor Henry M. Levy and the Opal group for their help and support during the preparation of this manuscript. In particular, Jeff Chase, with whom the author has spent many hours discussing the projects, and who provided valuable feedback on earlier versions of this manuscript. Ashutosh Tiwary and Mike Feeley deserve a special thanks for answering a lot of questions concerning the Opal implementation. Finally, thanks to Geeta Shukla who proofread the manuscript without having much understanding of the contents.

Chapter 1

Introduction

The client/server model is used to provide sharing and protection of data in many software systems. Using this model, distrusted applications (clients) can access and possibly modify shared data, while guaranteeing data integrity. To ensure this data integrity, the shared data is managed by a server that is located in a private protection domain. Client applications thus access the data through a fixed procedural interface defined by the server. Calls to the server domain utilize a protected procedure call mechanism, typically *Remote Procedure Calls*[Birrell & Nelson 84], to execute code within the server's domain. For example, an operating system's data structures are protected against unauthorized modification by the applications it is running. The applications can only modify the state of the kernel by invoking a trap that transfers the control of execution to a kernel-specified point in the trusted kernel code.

Structuring software systems according to the client/server model can improve extensibility, increase maintainability, and provide independence of failure. As long as the interface to the server is fixed, new clients and new implementations of the server can be made independent of each other. Also, clients can fail independently without affecting each other. The major drawback is that the model can suffer in performance, due to the domain-crossing costs required by the protected procedure calls.

This thesis describes *Porc*, a toolkit for building object-based client/server applications in C++. *Porc's* main goals are (1) to provide efficient access to protected objects and (2) to hide the protection boundary from clients. While parts of *Porc* existed before, this work concentrates on extensions of *Porc* to use read-only memory shared between client and server, in order to provide fast access to protected objects.

1.1 Object-based RPC

Porc consists of a runtime system and binding service that extends RPC to improve support for *object-based* services. For an object-based server, each RPC call pertains to a particular instance of an abstraction supported by the server. For example, an RPC call to a file server typically operates on only one of the server's files. Structuring server data as objects allows the toolkit to manage complex pointer-based data structures without having knowledge about the internal representation of each object. The toolkit only needs to know when pointers to protected objects[1] are being passed between a client and a server, because they have to be handled specially.

Object-based services have many needs that are not met by RPC alone. Each object must have a name or reference that a client can use to bind to the server and identify the object; access must be controlled on an object basis; objects must be reclaimed when clients no longer need them; some form of accounting for objects must exist. *Porc* is designed to support these needs on top of a standard RPC mechanism, i.e., it is built below the application but above the operating system's RPC facility. This approach contrasts with systems ranging from Hydra[Wulf 74], a capability-based system, to Mach[Accetta et al. 86], a micro-kernel operating system, which have built object support into the operating system kernel.

The toolkit simplifies the development of RPC clients and servers in the C++ program-

[1]Protected objects refer to objects managed by a server.

ming language. Protected objects are represented by local *proxy* objects[Shapiro 86], which make the RPC calls to the server. Proxies are heap-allocated and named by virtual addresses. Similarly on the server side, the details of receiving incoming RPC calls and converting them into ordinary procedure calls to the protected objects are handled by *guard* objects. Proxies and guards minimize the syntactic burden of RPC on both the server and the client side and efficiently hide the protection boundary from both.

1.2 Goals of the Toolkit

The goal of *Porc* is to permit richer uses of RPC, beyond today's access to system services that are statically bound by symbolic name. Dynamic binding and anonymous RPC interfaces permit callbacks and transparent binding to services. In particular, this can make protection easier to use by allowing modular programs to be split into different protection domains along module boundaries. Three issues are essential to accomplish these goals:

- Transparency of protected objects for clients,
- Security of protected objects for servers, and
- Synchronization of concurrent access to shared objects

Ideally, it should be completely transparent to the client code that some objects are protected and others are not. It should require minimal changes to the source code to add and remove protection boundaries during program development. The transparency is achieved by representing protected objects as proxies, thereby making them look like ordinary C++ objects. Pointers to proxies are, unfortunately, not completely interchangeable with pointers to local objects. Using proxies introduces pointer aliasing, and it is impossible to share proxy pointers between clients. Both of these problems occlude the transparency of protected objects.

Security means that server objects can only be modified by methods supplied by the server, so it can maintain data integrity. This is vital for securing independence of failure, i.e., to prohibit one client from corrupting the server and causing other clients to fail. Note that security implies protected access to modify server data, but not necessarily complete data hiding from clients.

Access to data managed by a server will inherently be concurrent, because multiple clients can be connected to a sever at the same time. A server must synchronize access to its objects. The synchronization must prevent two (or more) clients from updating a given memory location at the same time, possibly leaving an object in an inconsistent state, and it must prevent clients from reading data that are in the middle of being updated. This need for synchronization is the main reason why protection of objects cannot be done in a completely object transparent way.

1.3 Exploiting Shared Memory

Efficiency considerations are typically the main reason why many programs execute in a single protection domain, rather than in several protection domains. The cost of performing an RPC call and validating arguments in the guard is considerably more expensive than invocation of a local object. Bershad *et. al.*[Bershad et al. 90] have shown that the performance of RPC can be considerably improved by using shared memory in the local case where the sender and receiver are both on the same machine. Lightweight Remote Procedure Call uses a shared stack (A-stack) to avoid copying parameters and return values between protection domains.

This thesis investigates how efficient access to protected services can be further improved, in the local case, by removing the RPC call altogether, by taking advantage of shared read-only memory. The following three techniques are presented.

Shared Proxies

Each protected object has exactly one proxy which is located in the server and is callable by all clients. When a client binds to a server, the server maps its proxies read-only into the client's protection domain. No local proxy objects need to be constructed in the client's domain, making it faster to pass protected pointers around. An additional benefit is that there are no aliasing or sharing problems for proxy pointers.

Read-only Methods

Read-only methods are an efficient technique for reading data from a server. The server's data area is mapped read-only into the client's protection domain, thus allowing the client to access the objects directly. This eliminates the need for RPC calls and validation of parameters for querying calls. Modifications to the server's objects must be done with RPC calls, so the security of the server's objects is still maintained.

Version-based Synchronization

Synchronization of access to shared data turns out to be a problem with read-only methods, because they cannot directly acquire or release any server locks. A solution to this problem, based on a technique borrowed from lock-free synchronization, is presented.

Client/Server applications that employ these techniques need to coordinate their memory usage, to prevent the server's data area from overlapping with the clients. An implementation of *Porc*/C++ has been built for the Opal system[Chase et al. 92a, Chase et al. 93], a single virtual address space operating system. Read-only methods and shared proxies can be implemented very naturally in Opal's single address space, because there is no overloading of virtual addresses. None of the techniques described, however, depend on the single address space model, and can possibly be used on any operating system where applications can coordinate sharing of memory in some manner.

1.4 Related Work

Proxies were defined by Shapiro[Shapiro 86] as a structuring mechanism for distributed systems. They are local server representatives that hide the communication boundary from clients. In addition to being a structuring mechanism, Shapiro describes several other benefits, such as access control and caching of server data. *Porc* uses proxies to represent objects that are not in the client's protection domain to make protection boundaries transparent to the application programmer. Access control to protected objects is controlled on a per-object basis by access validation performed by a protected object's guard in the server. The idea of caching data locally in proxies to provide fast access to server data is generalized in *Porc* by the use of *read-only methods*, which let clients access server data directly without the need for RPC calls.

Object-based RPC has been used in numerous operating systems and distributed programming systems. Systems such as Hydra[Wulf 74], Eden[Almes et al. 85], Clouds[Allchin & McKendry 83], and Choices[Dave et al. 92] have all built object support into the operating system kernel. Objects are protected from each other with hardware protection boundaries. In Eden and Clouds, objects are represented as separate address spaces making them inappropriate for building applications with fine-grained sharing. Another approach is taken by systems such as Emerald[Jul et al. 88] and Pilot[Redell et al. 80]. They provide fine-grained sharing by ensuring protection through the use of a single safe language. *Porc* provides a combination of the two approaches by relying on hardware protection between applications and language protection within an application. The trade-off between hardware protection and performance are controlled by the application programmer.

The goal of Emerald is to create a system that efficiently supports fine-grain mobility of objects. To achieve high performance, all objects on a local machine are located in the same protection domain, so access to local objects can be done directly with load and store operations (as mentioned above, security is ensured by the use of a safe language).

Similarly, read-only methods in *Porc* provide fast read access to servers that are located on the same machine, but without the need for a safe language.

The proxy implementation in Choices is similar to shared proxies, but the rationale behind the design is different. In Choices proxies are allocated in read-only memory by the kernel to assure authentication of proxies and to be able to change the implementation of proxies without having to re-compile client programs, however, proxies are not shared between clients. Choices has an interesting table-driven RPC mechanism, where the RPC stub code is chosen dynamically on each call, which could for example be used to automatically select the use of read-only methods at run-time.

Several experimental operating systems, such as Monads[Rosenberg 92], Mungi[Heiser et al. 93] and Opal[Chase et al. 92a, Chase et al. 92c] provide a single shared address space for all applications. Protection and addressing are decoupled in these systems which makes it possible (among other things) for applications to easily coordinate sharing of data at run-time. The techniques for improving the performance of an object-based toolkit by using read-only shared memory presented in this thesis are inspired by these systems.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows. The next chapter describes the structure and implementation of the object-based RPC toolkit. Chapter 3 discusses the design and implementation of shared proxies. Chapter 4 explains the concept of read-only methods and their implementation. Chapter 5 describes version-based synchronization, a technique for read-only methods to efficiently synchronize access to protected objects. Chapter 6 evaluates the performance benefit of the techniques that are presented. Finally Chapter 7 presents the conclusion.

Chapter 2

Object-based RPC

Porc is a toolkit for building object-based client/server applications. The toolkit is used as a testbed for implementing and evaluating the performance improvements of shared proxies and read-only methods. This chapter gives an overview of the proxy model and the implementation of *Porc* in sufficient detail to understand the design and implementation issues of the techniques presented in this thesis.

The original toolkit was written by Jeff Chase and Mike Feeley. It was conceived as a way to extend Emerald's[Jul et al. 88] uniform object model to a traditional system with protection domains. It was developed as a sideline to the Opal[Chase et al. 92b] project, whose primary goal is to make protection easy to use, and to allow protection configuration decisions to be deferred as long as possible, through the use of a single address space architecture. *Porc* furthers this goal by allowing object-based applications to be separated into protection domains in arbitrary ways, with minimal effect on the source code. It also provides a uniform way to support rich RPC interaction between protection domains.

The goal is to build a uniform object system that uses two referencing forms: virtual addresses and capabilities, each optimized for the particular style of object. Local objects are named with virtual addresses, and protected objects are named with *capabilities*[Levy 84, Mullender & Tanenbaum 86]. A capability is an unforgeable reference that a client uses to name an object in another protection domain. Shapiro's proxy model is used to hide the non-uniformity, by encapsulating the capabilities in proxy objects, which in all respects act like the protected object. This approach gives application programmers almost everything they would get from capability hardware, only faster and on conventional hardware.

The following sections will describe in detail how the proxy model is used in *Porc*, how protected objects are named with capabilities, and how the toolkit interfaces to the underlying operating system's RPC mechanism. Only the aspects of *Porc* that are essential for understanding the design and implementation of read-only methods and shared proxies are discussed. Several other important issues, such as the type model, initial binding and reclamation of objects, are not discussed.

2.1 The Proxy Model

The proxy model is used to hide the protection boundary from the application. A high-level view of the model is shown in Figure 2.1. Protected objects are represented as proxy objects in the client's domain. A proxy marshals the parameters and executes an RPC call. On the server side, the RPC call is handled by a guard object, which unmarshals and validates the parameters and executes a local object invocation. The code for proxies and guards is ideally generated by a stub generator.

Proxies and guards correspond to client stubs and server stubs, respectively, in a standard RPC system[Birrell & Nelson 84]. Primitive values such as integers and text strings are marshaled and unmarshaled in the same way as for RPC. However, in addition, the proxy model allows pointers to protected objects to be passed between clients and servers. Pointers to protected objects are passed as protected object references (*capabilities*), so

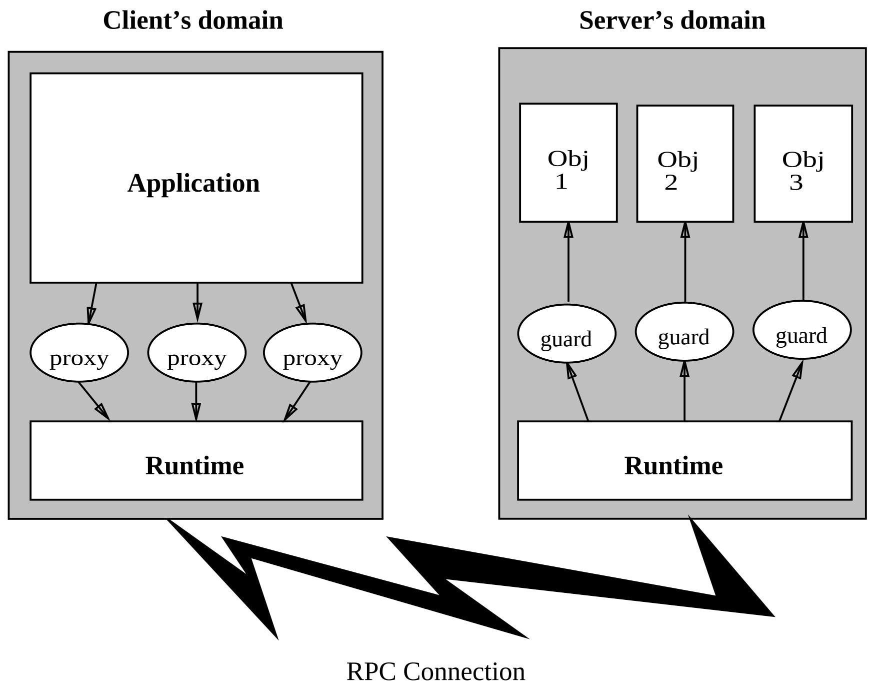

Figure 2.1: System Overview

both servers and clients can detect invalid pointers and avoid dereferencing them.

Each protected object has at most one guard, which among other things contains the capability for the protected object. Passing a protected pointer to a client involves creating the guard object (if it does not already exist), retrieving the capability from the guard, and handing the capability to the underlying RPC mechanism. On the client side, the proxy that receives the RPC call translates the capability into a proxy pointer. Translating a capability consist of looking it up in a hashtable that maps capabilities to proxy pointers. The proxy pointer is directly returned if the capability is found in the hashtable. Otherwise, the client instantiates a new proxy for that object, stores a pointer to it in the hashtable and returns the proxy pointer.

Passing a protected pointer from a client to a server is slightly different. Protected pointers are passed as capabilities, which also are stored in the proxies. The guard that receives the RPC call statically knows whether the capability refers to an object that is local to the server or not. For capabilities that refer to a local server object, the object can be found directly by examining information stored in the capability, thereby avoiding a hashtable look-up. A capability that refers to a non-local object is handled exactly as explained above, i.e., by looking the capability up in a hashtable.

Porc almost completely hides the protection boundary by employing the proxy model, however, it introduces pointer aliasing. Pointer comparison cannot be used to detect object identity, because there can be several proxies that refer to the same protected object.

Finally, how do we get the first proxy? Initially, a client has one or more proxies for objects exported by the server. These proxies could be read from shared or persistent storage, or they could be built locally from capabilities retrieved from a name server.

2.2 Capabilities

Capabilities are unforgeable objects used to name protected objects. A capability gives the owner the right to bind to the protected object and invoke its methods. *Porc* uses *password capabilities*[Anderson et al. 86, Chase et al. 92c] to uniquely name protected objects. A password capability is probabilistically rather than absolutely impossible to forge. Each capability has a (64-bit) password associated with it. The run-time system checks the password on each object invocation to authenticate the request. Capabilities are supported independent of the particular RPC transport, allowing portability across operating system platforms and compatibility with optimized RPC implementations. In particular, *Porc* can be used on systems with no explicit support for capabilities.

The contents of a capability are shown in Figure 2.2. The *portal* identifier is a unique 64-bit value that functions as a global name for an RPC end-point. From this value a client can obtain an RPC connection to the server. This will be explained in more detail in the next section. The *object* identifier uniquely names an object within a server, and

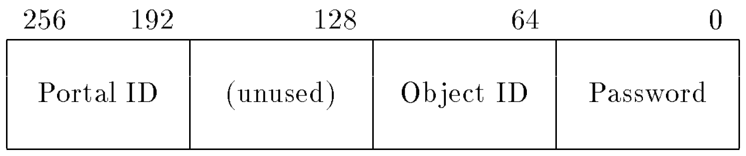

Figure 2.2: Contents of a Capability

the *password* is the randomized check field.

The capability is minted by the guard for a protected object. The guard knows the portal ID of the RPC channel on which it is receiving its calls. The object ID is used to identify the guard object. This object ID could be anything, for example, an index into a table of guards. In the current implementation it is simply the virtual address of the guard. This pointer must itself be validated before it is followed. This is done with a magic number stored in each guard. The password is randomly assigned by the guard and stored in both the capability and locally in the guard.

Security for protected objects relies on access control checks within the server after a request has arrived. The guard object for the protected object is located using the object ID in the received capability. The guard then validates the capability by comparing the password field with the value it has stored before calling the protected object. In general, the server-side application code is not aware of the identity of its clients — or even that the call originated from another protection domain. Thus the capability checks are critical for safe functioning of the system.

2.3 Portals and Channels

Objects are named uniquely within a server by their object identifiers. System-wide names are achieved by identifying each service with one or more portal identifiers, which are global names for RPC end-points. *Porc* includes a location broker service, the *Portal*

Service, that assigns unique numbers for portals and maintains a mapping from portal IDs to RPC bindings. Applications can call the portal service to obtain an RPC binding for a specific portal.

Porc is designed to use any message-passing or RPC mechanism as a transport, with the caveat that marshaling of protected pointers must be handled specially as explained above. The underlying transport is hidden in two C++ classes. The *PorcCall* class represents a call or reply message buffer, and the *Channel* class represents a client-side message connection to a particular portal. The interfaces to these classes are fixed, but the contents and implementation are transport dependent. For example, in the Mach-based implementation a Channel caches the send-name of a Mach[Accetta et al. 86] port to the server. In an LRPC-based implementation[Bershad et al. 90], the Channel would hold the portal ID and a pointer to a memory region (A-stack) which is shared with the server.

For efficiency, Portal-ID-to-Channel bindings are cached in a hashtable (the *PortalID table*) locally in each client to avoid a call to the PortalService each time a Portal ID needs to be converted to a Channel object. The Portal Service is only called when a new server is instantiated or when a client imports a capability from a name server. In the first case, the server is assigned a unique portal number, and in the second case, the client needs to obtain an RPC connection to the server. A client has one Channel for each server it is connected to. Each proxy caches a pointer to the Channel for its server, so it can be located quickly. This is essential for performance, but unfortunately has the disadvantage that proxies cannot be shared between clients. The RPC mechanism hidden by the Channel is usually domain-specific, so Channels can therefore only be used by the domain that created them.

2.4 Summary

This chapter provides an overview of the key ideas behind *Porc*. Protected objects are located in designated servers; distrusted clients name protected objects with capabilities, and call them using standard RPC as a transport; protected calls indirect through proxy and guard objects whose methods are marshaling stubs; proxy objects cache the channel to its server; and the guard objects uses a password field to authenticate access before directing incoming calls to their associated objects.

The transparency of protected objects is occluded in several ways in the version of *Porc* described above. First, it is not safe to compare pointers to detect object equality, because two different instances of a proxy can reference the same protected object. Second, clients cannot pass proxy pointers in shared memory, because proxies contain domain-specific state. Third, protecting objects might be undesirable, because of the overhead involved with binding to protected objects and invoking their methods. The next two chapters will describe two concepts, read-only methods and shared proxies, that address these problems.

Chapter 3

Shared Proxies

The toolkit as explained in the last chapter provides a programmer with the basic mechanisms for implementing object-based clients and servers. Protected objects are completely separated from the client and can only be accessed indirectly through local proxies. The proxy model is used to hide the protection boundary from the client, making protection nearly transparent.

However, the proxy model introduces extra overhead when accessing protected objects. When a server passes a capability (a protected pointer) to a client, it has to be converted into a pointer to a local proxy. This involves (1) looking up the capability in a hashtable and (2) possibly instantiating a new proxy. Creating proxies locally in each client also prevents that proxies from being shared between clients. For example, if a server contains a hierarchical structure of files, and a client wants to calculate the total number of bytes used to store the files, a local proxy is instantiated in the client for each single file in the server. If another client decides to do exactly the same operation, it cannot reuse any of the proxies already created, but again has to create a local proxy for each protected object (Figure 3.1).

Shared proxies are designed to avoid these problems. A shared proxy is created by the

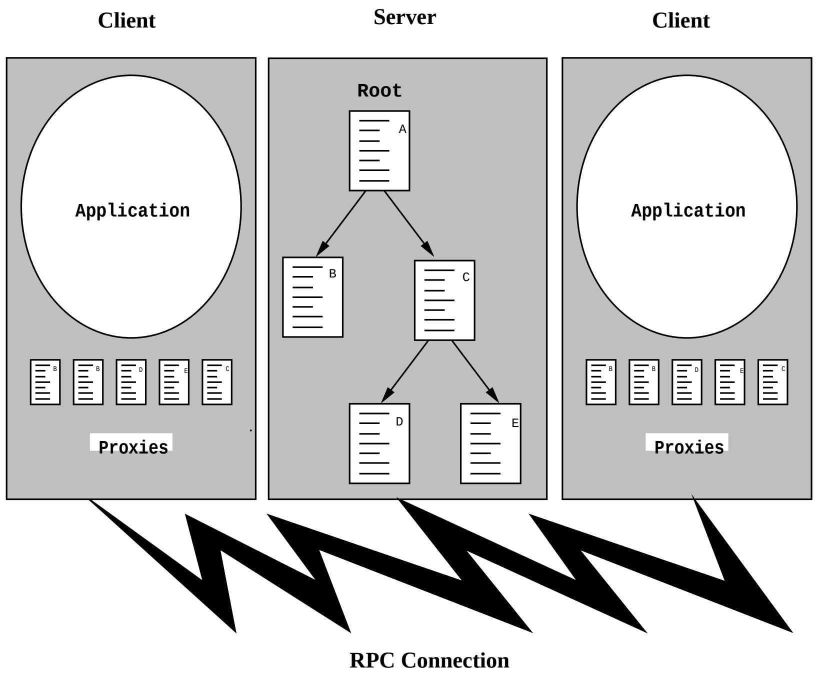

Figure 3.1: An object-based file server

server when a new protected object is created and it is located in the server's protection domain. When a server exports a reference to a protected object, it maps the proxy read-only into the client's protection domain and returns a pointer directly to the shared proxy. The key concepts behind shared proxies are:

- Exactly one proxy per protected object. A shared proxy is created by the server and located in the server's protection domain. This contrasts to normal proxies, where there is one proxy per protected object per client. The server can directly pass a proxy pointer to the client, thereby avoiding a hashtable look-up on the client side.
- Callable by all clients. A proxy might be used by multiple clients at the same time, so it dynamically chooses which Channel object to use on each call.
- Read-only. Shared proxies are mapped read-only into the client's domain to main-

tain the security of the system. All clients share the same proxy, so if a client could intentionally or unintentionally corrupt a proxy that might cause other clients to fail.

Attachment of shared proxies to clients is done eagerly, i.e., when a server exports a reference to one of its protected objects, it attaches the shared proxies for that object and all other objects that the client is likely to access. This makes future accesses to objects in the server faster because proxies are already attached, and amortizes the cost of attaching proxies over many server invocations. The cost of attaching segments to a client is high compared to the cost of allocating memory, because it requires a system call. For the file server example above, when a client has access to the root-directory, it can potentially access all of its sub-directories. Therefore, all the proxies for the directories can be mapped into the client's domain when the root-directory is initially accessed. Another example where shared proxies are useful is when a new domain is instantiated in Opal[Chase et al. 92b]. The Opal server can instantiate proxies for all its system services, simply by attaching a single segment containing all the shared proxies to the new domain.

An additional benefit of shared proxies is that pointers to them are almost completely interchangeable with pointers to local objects. There are a few subtleties with shared proxies. A mechanism for efficiently locating channel objects dynamically on each call on a shared proxy must be devised, and binding to shared proxies is different than binding to ordinary proxies. These issues will be discussed in the following sections.

3.1 Transparency

One of the goals of *Porc* is to make protection of objects transparent to clients, so protection boundaries can be inserted and removed with minimal source code modifications. For example, it should be possible to insert extra protection boundaries when debugging

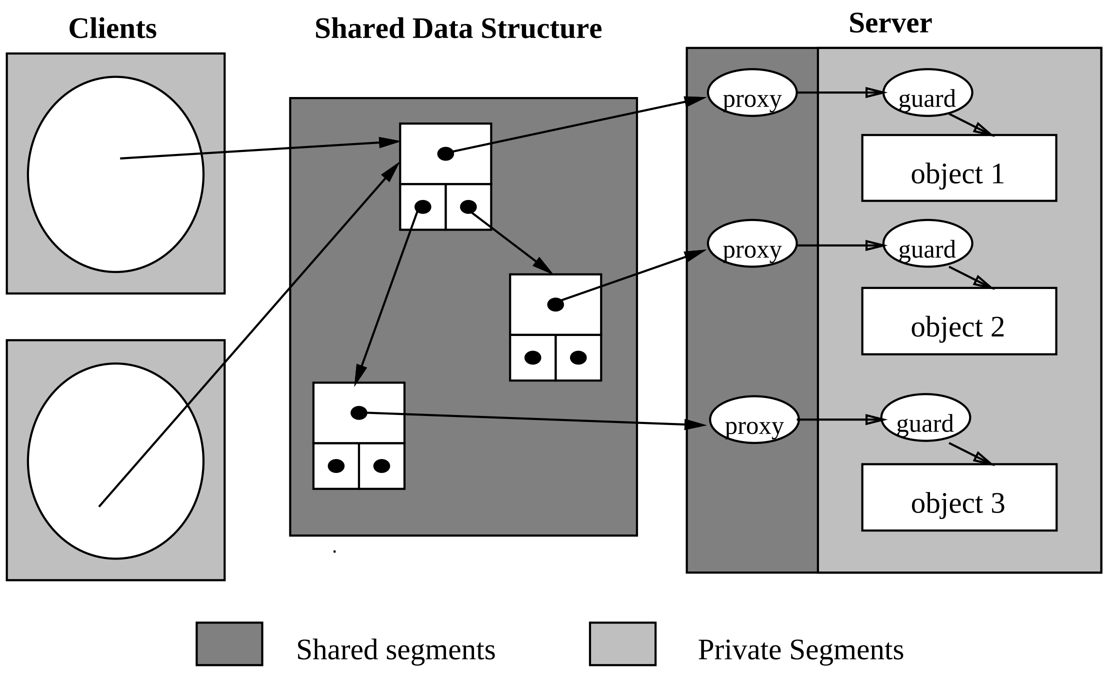

Figure 3.2: Example of a shared data structure

large applications to track down run-away pointers.

The implementation of *Porc*, described in Chapter 2, introduces pointer aliasing and does not allow pointers to protected objects to be shared between clients. This occludes the transparency of the toolkit.

Shared proxies eliminate the problems with pointer aliasing and domain-specific proxies. Each protected object has exactly one proxy, so no pointer aliasing occurs. Furthermore, proxies are callable by all clients, so pointers to shared proxies can be stored in shared memory and can be used by any client. Figure 3.2 shows an example of two clients sharing a data structure containing pointers to protected objects.

3.2 Channel-less Proxies

Channel objects, which represent a communication channel between a client and a server, are inherently domain-specific, because *Porc* is designed to use the native RPC mechanism supported by the underlying operating system. A shared proxy can be used by an arbitrary number of clients at the same time, so it has to dynamically select the Channel to use on each call. This contrasts with a non-shared proxy, which only has to cache a reference to a single Channel.

The PortalID table caches all the channel objects that connect a client to its servers. The Channel a shared proxy must use to communicate with the server can therefore be retrieved from the PortalID table that belongs to the client that is executing the call[1]. Channel objects are looked-up by their portal ID. The capability for the protected object is already stored in the proxy, hence, the proxy has direct access to the Portal ID of the server.

Security is maintained, even though shared proxies are accessing unprotected PortalID tables, because proxy code is executed by clients and a client can only access its own PortalID table. A corrupted PortalID table will not affect the server or other client applications. Reclamation of Channels when a client terminates trivial, because they are stored locally in the client, and the complexity of locating a Channel solely depends on how many channel objects the calling client itself has stored.

The overhead of making calls on shared proxies is necessarily higher than on ordinary proxies, because a Channel has to be looked up on each call. Access to an object-based server is expected to often consist of following pointers from one protected object to another protected object in the same server, i.e., a client thread will, with high probability, use the same Channel as it used on the previous call on a shared proxy. The look-up scheme has been optimized for this situation by caching the most recently used

[1]How the calling client's PortalID table is located will be described in section 3.4.

Channel in a thread's control block. The added overhead of looking up channel objects is expected to be small compared to the overhead of making RPC calls that is already present.

3.3 Binding

Normally, when a client binds to a protected object the server passes it a capability; the client uses the capability to instantiate a local proxy and to retrieve an RPC binding to the server. For shared proxies, the proxy already exist in the server. It has to be attached read-only into the clients domain and the client must retrieve a pointer to it from the server.

Binding to protected objects with shared proxies is very efficient in the common case, where the server returns a reference to one of its own objects. In this case, the server directly returns the shared proxy pointer to the client. The client is required to trust the server, because it cannot verify the pointer in any (straightforward) manner, but it avoids a relatively expensive hashtable look-up to translate a capability.

A client still has to pass pointers as capabilities to a server, so they can be validated. The server can directly convert a capability for one of its own objects to a pointer by decoding the *Object ID* field in the capability. However, in the situation where a server receives a capability for a non-local object (or an application is receiving a capability from a name server) a method for binding to the shared proxy, given a capability, is needed.

Porc handles this situation by implementing a hidden bootstrap method for each protected object with a shared proxy. The bootstrap method, when called, maps the shared proxy read-only into the caller's protection domain and returns a pointer to it. The guards know statically if a capability refers to an object with a shared proxy, so they can automatically generate an RPC call to retrieve the shared proxy pointer. This method requires an extra RPC call, but it is believed to be an uncommon case that a client passes capabilities for non-local objects to a server.

The performance increase that shared proxies provide is in part achieved by eagerly attaching a group of proxies into a client's protection domain when the client initially imports a reference to a single protected object. Proxies are by default organized into groups by type, i.e., all proxies that refer to objects of the same type are attached to a client's domain together. Several types can also be grouped together, so they will be attached at the same time. For each group the toolkit maintains a list of which clients have got proxies attached to prevent redundant attaches.

A granularity of a type may be too coarse for certain applications, which need to have finer control over which objects a server has access to. The group an object belongs to can therefore also be controlled on a single object basis, but that requires extra code by the implementor.

3.4 Implementation

Three extensions need to be made to *Porc* in order to support shared proxies: (1) creating shared proxies in the server, (2) a way for shared proxies to locate a client's local PortalID hashtable, and (3) attaching shared proxies to clients.

A shared proxy is created along with the guard. Shared proxies are by default created lazily when the first reference to a protected object is exported, but they can also be created eagerly by proxy code. Figure 3.3 shows how a server keeps track of proxies and guard objects. The protected object contains a pointer to both the guard and the shared proxy; the guard and the proxy both contain a pointer to the protected object. There is also an RPC connection (Channel) between the proxy and the guard. The pointer from the protected object to the shared proxy is used by the guard stub code for converting a real object pointer to a shared proxy pointer.

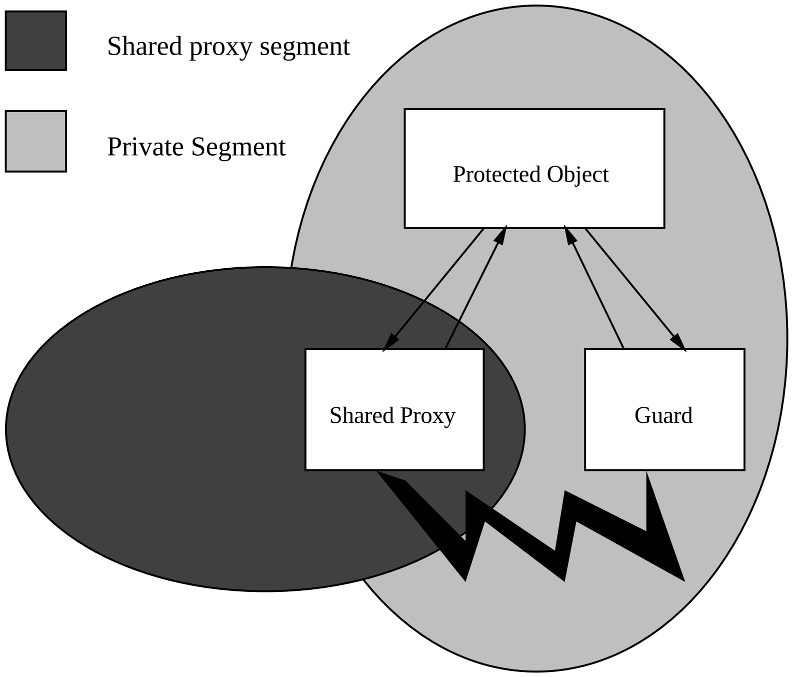

Figure 3.3: Shared proxies, guards and protected objects

Stub code for the guard needs to find the PortalID table that is local to the client that executes the call. A semi-independent thread package[Feeley et al. 93] that comes along with the toolkit is modified to support this. In the thread package, a thread's stack is power of two aligned and of fixed size. Given a stack pointer, the top of the stack can be found by masking off a constant number of bits, and the first word of each stack contains a pointer to the *thread control block* (TCB). Each running thread can therefore access its TCB by masking off a number of bits of the current stack pointer. The TCB has been modified to hold a pointer that caches a Channel and to hold a pointer to the client's PortalID table. A method *getChannel* is added to the thread package, which returns the Channel for a given portal ID. The method first checks is the Channel is cached in the TCB and in that case directly returns the cached pointer, otherwise it resolves to a hashtable look-up into the PortalID table. Figure 3.4 shows a code fragment for a shared proxy.

```
NameProxy::set(char *name)
{
   Channel *cnn = thisthread->getChannel(portal_id);

   ... Marshal parameters and executes RPC call ...
}
```

Figure 3.4: Stub code for a shared proxy

The final change to the implementation of *Porc* is to attach a server's proxy segments read-only into clients' protection domains. Opal's single address space greatly simplifies this, because it assures that all applications are located in separate non-overlapping address ranges, and it also provides a mechanism for mapping segments from one protection domain into another protection domain. A server might need to attach segments to a client each time a capability to one of it's objects is exported. For those methods *Porc* silently passes an *attach-segment*-capability for the client's domain to the server, so it can attach the proxy segment read-only to the client . The server stores the client's *attach-segment*-capabilities in a hashtable to prevent further redundant attaches.

3.5 Summary

This chapter introduced shared proxies and explained how they can improve performance of the *Porc* toolkit. Shared proxies make binding to protected objects potentially faster by mapping shared proxies read-only into client's protection domains in bulk, thereby avoiding allocation of memory and initialization of proxy objects locally in clients; they allow the server to pass proxy pointers directly to the client, avoiding the cost of translating between virtual addresses and capabilities; and a higher degree of transparency is achieved, because they do not introduce pointer aliasing and pointers to shared proxies can be shared by multiple clients.

However, calls on shared proxies are inherently slower than calls on ordinary proxies, because the Channel must be located dynamically on each call. A scheme that caches the most recently used channel in each thread has been devised, so only one extra indirection is needed to find the channel object in the common case where a thread calls the same server multiple times in a row. The overhead of locating the channel object is expected to be low in comparison to the performance gains from eliminating proxy instantiation and conversion of protected pointers. An evaluation of shared proxies is presented in Chapter 6.

Chapter 4

Read-only Methods

Read-only methods provide a technique for optimizing read-only access to protected objects. Like shared proxies, the scheme is based on the use of shared read-only memory for improving the performance of object-based client/server communication.

Access to protected objects is inherently more expensive than access to a local object, due to the cost of the RPC call to the the server's protection domain. Even on highly optimized RPC protocols such as LRPC[Bershad et al. 90] it will be considerably more expensive. The extra overhead for an RPC call consists of marshaling the arguments and return values, two kernel invocations to transfer the marshaled arguments between the domains, and validation of the arguments in the guard.

The context-switch can be avoided for functions that do not modify the state of the server, but only read data from it. We will call these *read-only methods* or *r-methods*[1]. The idea is to give the client read-only access to the server's data, so protected objects can be queried as efficient as local objects. Figure 4.1 shows an example of a client that has read-only access to a server. The client has pointers directly to data structures managed by the server and can follow pointers stored in the server directly without

[1]In contrast to *write-methods* or *w-methods* that modifies the state of a server

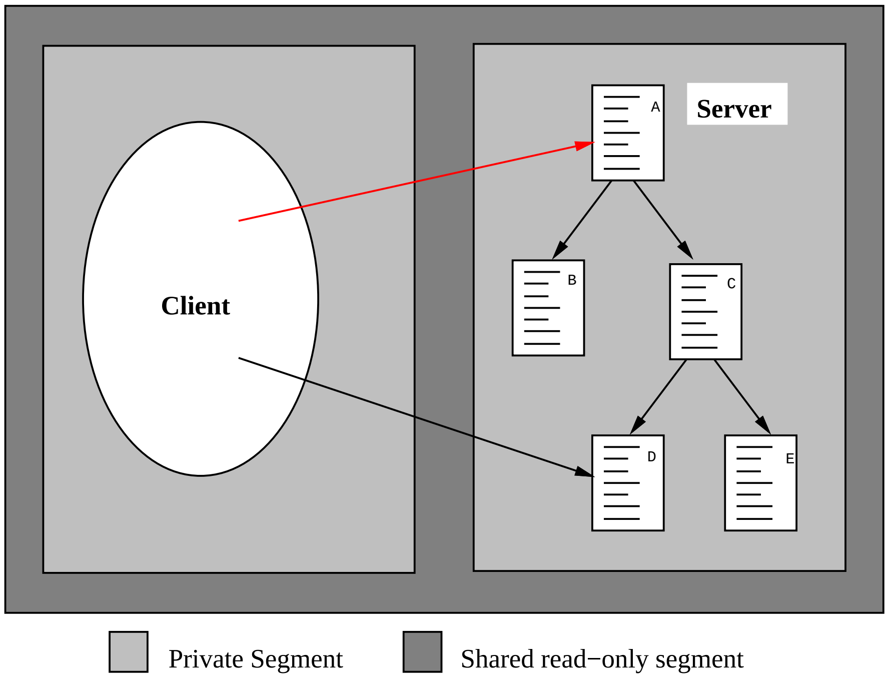

Figure 4.1: A client with read-only access to a server

having to translate them.

The key concepts of read-only methods are:

- A server attaches its data and code segments read-only to a client's protection domain, thereby allowing the client to directly query its objects.
- The security of the server is maintained. Clients only have read-only access to the server and therefore cannot directly modify any of its objects. Modifications must be done with RPC calls to the server's domain.
- They are completely transparent to the client application. Read-only methods are implemented as proxy methods. However, parameter validation is relaxed, because they are executed by the client, leaving the server unaffected even if a bad pointer is followed.

There are certain pitfalls the server implementor has to be aware of when implementing read-only methods. Granting clients read-only access to the server implies that the server does not have any control over which objects are being accessed. Only the objects that a client is allowed to access should be attached read-only to the client's domain. The segments that get attached to the client should also not contain any information that makes it possible for a client to synthesize fake capabilities. Synchronization of concurrent access to objects with r-methods causes a problem, because r-methods cannot directly modify locks in the server. The next chapter is devoted to that problem. Despite these problems, r-methods are perfect for achieving high performance for applications that are divided into multiple protection domains to increase extensibility and provide independence of failure.

In Shapiro's original proposal of the proxy model he describes a method for optimizing client/server communication by caching immutable data in proxies, thereby avoiding a context-switch. Read-only methods generalize this idea by accessing the data directly in the server. They have several advantages over caching of data in proxies. By storing data only in the server, it is possible to change data and all the clients will see the change instantly. Another benefit is that the cost of copying data into a proxy is avoided.

The next section describes how parameter validation differs for read-only methods, and section 4.2 discusses how read-only methods support is implemented in *Porc* and how stub code can be generated for them.

4.1 Parameter Passing and Validation

A key mechanism for ensuring security for the servers is the pointer validation done by the guards. Pointers must be passed as capabilities, so the server can make sure that a pointer is valid and actually pointing to a protected object of the type the server expects. This prevents the server from following an invalid pointer.

Read-only methods are implemented as proxy methods, but instead of executing an RPC call, they make an ordinary invocation of the corresponding method on the protected object in the server. Validation of arguments and return-values for r-methods can be relaxed, because they are executed by the client. If a client tries to follow an invalid pointer it can only hurt itself. The server will always remain unaffected. In particular, pointers can be passed directly to r-methods without translating them to capabilities, making it possible to access complex pointer structures efficiently, without sacrificing security.

However, pointers need to be translated between proxy-pointers and protected-object-pointers in the stub code for r-methods. All pointers that are manipulated by a client are proxy pointers, so the stub code for read-only methods must convert (proxy) pointers passed as arguments to pointers that point directly to the protected object, and vice-versa for pointers that are returned as a result. The proxy-pointer is translated by looking up the protected-object-pointer in the proxy. Translating pointers returned from the server is a little more complicated. If a protected object uses ordinary proxies, the r-method needs the capability from the guard to find or instantiate a proxy in the client's protection domain. Otherwise, if the protected object has a shared proxy, then the pointer to the shared proxy can be returned. In either case, the guard (and the possible shared proxy) must exist, because an r-method cannot modify the server. Guards and shared proxies must therefore be instantiated eagerly.

Read-only methods can also return pointers directly to a protected object, so a client can access the object directly without the indirection through the proxy. This can be useful when a server wants clients to have very efficient access to linked data structures. The disadvantage is that clients cannot modify the object a pointer refers to, because the client does not have access to the proxy pointer. However, returning pointers directly to protected objects could, for example, be used for a server that caches WWW pages. These pages are never changed by clients, and contain links to other pages that can be followed directly.

```
ObjectProxy *ObjectProxy::potato(ObjectProxy *p)
{
   Object *real_p  = p->GetObjectPtr();

   Object *real_r  = GetObjectPtr()->potato(real_p);

   return real_r->GetProxyPtr();
}
```

Figure 4.2: Sample proxy code for a read-only method

4.2 Implementation

This section describes how stub code is generated for read-only methods, and how the standard *Porc* implementation is extended to support read-only methods. Two key additions are made: initialization of a pointer to the protected object in proxies, and support for attaching server segments to clients.

Generating proxy code for read-only methods consists mainly of translating proxy pointers. Figure 4.2 shows the stub code for a function that takes a pointer to a protected pointer as an argument and then returns a pointer to another protected object. The method *GetObjectPtr* on a proxy object returns a pointer to the protected object, and the method *GetProxyPtr* on a protected object returns a pointer to the proxy for that object. Returning the proxy pointer consists of either returning the shared proxy pointer or of looking the proxy up in the PortalID table, depending on if shared proxies are used or not.

In order for a proxy to return a pointer to the protected object when the *GetObjectPtr* method is called, a new instance variable, *ObjPtr*, is added to each proxy, which points to the real object's locations in the server. The variable is initialized when the proxy is created, i.e., each time a new capability is passed to a client or server as a result of executing a method on a protected object. The capability does not contain the specific location of the protected object within the server, so that information must either be passed along with the capability or requested from the server after the capability has been received.

Clients are required to trust servers, therefore a server passes the pointer to the protected object with the capability. The pointer is stored in the proxy when it is instantiated. In the case where clients pass protected object pointers to a server, this scheme cannot be used, because a server does not have any way of validating a passed pointer. Instead, a mechanism where a server, given a capability, can make a protected call to the server to obtain the location of the protected object are implemented. This requires an RPC call, but it avoids the need for servers to trust pointers passed by clients. The extra protected call is executed by the guard objects and are completely transparent to applications.

A server might need to attach segments to a client each time a capability to one of its objects is passed. This is similar to attaching shared proxies into a client's domain and is implemented as explained in Section 3.4.

4.3 Summary

Read-only methods allow clients to obtain data directly from protected objects, avoiding the overhead of RPC calls and without sacrificing protection. As a side-effect, parameter validation is relaxed, so pointers can be passed directly as parameters or returned from r-methods. The underlying mechanism for supporting read-only methods is read-only sharing of memory between the client and the server. The next chapter will discuss how the synchronization of read-only methods can be handled.

Chapter 5

Version-based Synchronization

The previous chapter introduced read-only methods, which permit clients to efficiently read data from a server. Read-only methods eliminate the context-switches that transfer control between the client's and the server's domain. The client has read-only access to the server's domain, thereby allowing it to directly execute functions that only read data from the server.

An r-method cannot modify any data in the server, because otherwise a protection fault will occur when it is executed by a client. This causes a problem with synchronization of concurrent access to protected objects.

Synchronization of shared data is typically done with locks. A thread which is about to modify shared data acquires a write-lock before it applies any changes. This lock assures that it is the only thread accessing the data. After the modifications have been applied it releases the lock again, so other threads can access the data. Threads that are only reading shared data, i.e., executing an r-method, require a read-lock, so the data it not modified while they are reading it. In contrast to a write-lock, which only allows one thread to execute at a time, a read-lock allows several r-methods to execute at the same time.

Read-only methods cannot directly modify the server's locks, so the traditional mechanism described above can therefore not be used. Two ways of solving the synchronization problem are presented in this chapter: the obvious way of acquiring locks using RPC calls to the server and a scheme based on version numbers, a technique borrowed from lock-free synchronization[Herlihy 91].

5.1 Using Locks

For certain kinds of servers it might be profitable to arrange access to data in a check-in/check-out manner. A client makes an RPC call to the server (through a proxy) to check-out an object it wants to access. The check-out call requires a lock in the server for the specified object. Read-only access to the object can then be done by r-methods without any need for synchronization. When a client is done with an object, it checks it in, which releases the lock. For example, an object-based file system could be structured this way. Read-only access to files can then be implemented without any copying of the files. A simple example of a C++ class that uses this method is shown in Figure 5.1.

The server has to protect itself against ill-behaved clients that check-out objects and terminate. This could, for example, happen if a client gets prematurely terminated by the user with a `kill` command. The objects that are checked out will never be checked in, leaving them inaccessible forever. A simple mechanism to prevent this is to associate a timer with each server lock. For a client to maintain a lock it has to periodically make calls to the server to renew its ownership. The refresh calls can be hidden in the *Porc* run-time, thereby being completely transparent to the application.

Acquiring and releasing locks with RPC calls to the server is expensive. For the scheme to be faster than just executing a single RPC call to retrieve the data, the cost of the check-in/check-out calls must be amortized over cheap r-methods calls. The scheme is therefore best suited for servers with large objects and coarse-grained sharing.

```
const int MaxNameLength = 32;

class Name {
   Mutex mutex;
   char  name[MaxNameLength];
public:
   void checkIn()    { mutex.unlock(); }
   void checkOut()   { mutex.lock();   }

   void setName(char *);
   void getName(char *);                // Read-only method
};

void Name::setName(char *n)
{
   CheckOut();
   strncpy(name,n,MaxNameLength);       // Copy data into object
   CheckIn();
}

void Name::getName(char *n)
{
   strncpy(n,name,MaxNameLength);       // Copy data out of object
}
```

Figure 5.1: A simple concurrent C++ class using the check-in/check-out scheme

5.2 Version Numbering

Version numbering of objects is another scheme for r-methods to synchronize access to shared data. In contrast to the check-in/check-out scheme, it is best suited for servers with small objects and fine-grained sharing.

Instead of requiring a lock to make sure that the data is not modified, the technique is based on detection of modification and recovery. The server marks an object inconsistent during an update. Read-only methods can detect if an object has been marked inconsistent while it was executing, and in that case it can restart itself. The assumption is that the data accessed by an r-method is rarely modified during its execution.

The method relies on two version-numbers, CHECK[0] and CHECK[1], that are added to each object. If the two counters have the same value, the object is in a consistent state, otherwise it is not. A w-method marks an object inconsistent by incrementing the value of CHECK[1] before its applies any modification to it. When the update is done, the object is marked consistent again by incrementing CHECK[0]. The code for an r-method stores the value of CHECK[0], copies the data out of the object, and then compares the stored value against CHECK[1]. If the two values are equal then the method returns successfully, otherwise the method is restarted. A read-only method is guaranteed to detect if its execution overlaps that of a w-method, because the r-methods and w-methods access the version-numbers in reverse order. An example of how the technique is used is shown in Figure 5.2. Note that synchronization for w-methods is still done with locks.

The version numbers can be hidden by the subclass mechanism in C++. Figure 5.3 shows the interface for a class that encapsulates the version numbers and provides high-level methods to manipulate them. Objects that use version-based synchronization are derived from this class. Read-only methods read the value of the two version numbers using *lf_first* and *lf_last*. Write-methods use *lf_inconsistent* to mark an object inconsistent

```
const int MaxNameLength = 32;

class Name {
   Mutex mutex;
   long  check[2];
   char  name[MaxNameLength];
public:
   void setName(char *);                // Update method
   void getName(char *);                // Read-only method
};

void Name::setName(char *n)
{
   Mutex.lock();
   check[1]++;
   strncpy(name,n,MaxNameLength);       // Copy data into object
   check[0]++;
   Mutex.unlock();
}

void Name::getName(char *n)
{
   long first, last;

   do {
      first = check[0];
      strncpy(n,name,MaxNameLength);    // Copy data out of object
      last  = check[1];
   } while(first!=last);
 }
```

Figure 5.2: A simple concurrent C++ class using version numbering

class LockFreeObject		
`long`	`lf_first`	`()`
`long`	`lf_last`	`()`
`void`	`lf_inconsistent`	`()`
`void`	`lf_consistent`	`()`

Figure 5.3: Interface for the LockFreeObject class

before they apply any modifications, and call *lf_consistent* when the changes have been applied. The class also interacts with a class built to provide memory management, which are explained in the next section.

The efficiency of this scheme depends on how often r-methods are restarted, which in turn depends on the execution time of r-methods and the update rate of objects. The scheme is therefore best for objects where reads are more common than writes, and where r-methods execute quickly. A potential problem with read-only methods are starvation. For long executing r-methods or r-methods that access objects that are modified often, the probability of getting restarted is high. The execution-time of an r-method can therefore be arbitrary long. This can be avoided by having an upper limit on the times an r-method can be restarted; if that limit is exceeded an ordinary RPC call to the server is made to execute the function.

5.3 Memory Management

Memory management for objects that are accessed by read-only methods is special because memory for deleted objects cannot be freely reused, but can only be used for objects of the same type. A client cannot tell if an object it invokes by a read-only method has been deleted or not. If it is the case that the object has been deleted and its memory is reused for another object, then it is very likely that the read-only method will return garbage data, because the memory region now contains a different bit-pattern. What is even worse, is that a read-only method might be restarted infinitely, if the memory locations where it expects to read the version-numbers do not have the same values.

However, recycling the memory region to objects of the same type is possible, because they have the same memory layout. The drawback is that an r-method might return unexpected data, because the object it was accessing got deleted and replaced with another that belongs to a different data structure. How to handle this problem depends on the data structure. For example, for a linked list, the *first* and *next* primitives can be implemented so the effect of the above problem is that some elements will be accessed twice when the list is traversed. Another solution is to include an object identifier in each object, so an r-method can detect that the object has been deleted and return an error code. Appendix A shows how a singly linked list can be implemented with version-based synchronization, and explains the implications of the memory management in detail.

To manage the reuse of memory, each object type has a free-list associated with it. It is a linked list of pointers to memory regions that has contained objects of the specific type. When an object is deleted it is put unmodified on the free-list, so r-methods that are currently accessing it can still proceed, unaware that the object has been deleted. Memory for a new object is allocated by first checking the free-list and then if that is empty allocate memory directly from the heap. The new object is marked inconsistent while it is initialized to prevent r-methods from reading inconsistent data. However, managing a separate free-list for each type of object introduces memory fragmentation. A server might not be able to allocate memory for a new object, even thought there is a lot of unused memory, because the free memory region can only be for objects of another type.

The management of free-lists and allocation and deallocation of memory is supported by the *LockFreeMemPool* class. The standard `new` and `delete` operators in C++are overloaded, so memory is allocated and deallocated through the use of an instance of the *LockFreeMemPool* class. The interface for the class is shown in Figure 5.4.

class LockFreeMemPool		
`LockFreeObject*`	`allocate`	`()`
`void`	`recycle`	`(LockFreeObject *)`

Figure 5.4: Interface for the LockFreeMemPool class

5.4 Summary

This chapter describes the problem of synchronizing concurrent access to shared data for read-only methods and presents two solutions. One is based on a check-in/check-out mechanism, which is suitable for large objects when sharing is coarse-grained. The other solution, based on version numbers, is suitable for small fine-grained shared objects. In the common case, where data is not modified, version-based synchronization provides read-only methods with a low overhead mechanism for synchronizing concurrent access to shared objects.

Chapter 6

Evaluation

The previous chapters have presented shared proxies, read-only methods and version-based synchronization. These are techniques developed for improving the performance of object-based RPC in the local case where both the client and the server reside on the same machine. This chapter will evaluate the techniques and present performance results.

A name server is used as a test program. Four different versions of the name server are built using different combinations of shared proxies and read-only methods to analyze the performance improvements of the techniques both individually and combined.

The rest of this chapter describes the implementations of the name server in detail, presents the benchmark and evaluates the results.

6.1 The Name Server

The name server offers its clients a general purpose name-to-capability mapping. The name space is structured as a hierarchical collection of directories. A name is stored in a directory and contains either a reference to a capability or a reference to another

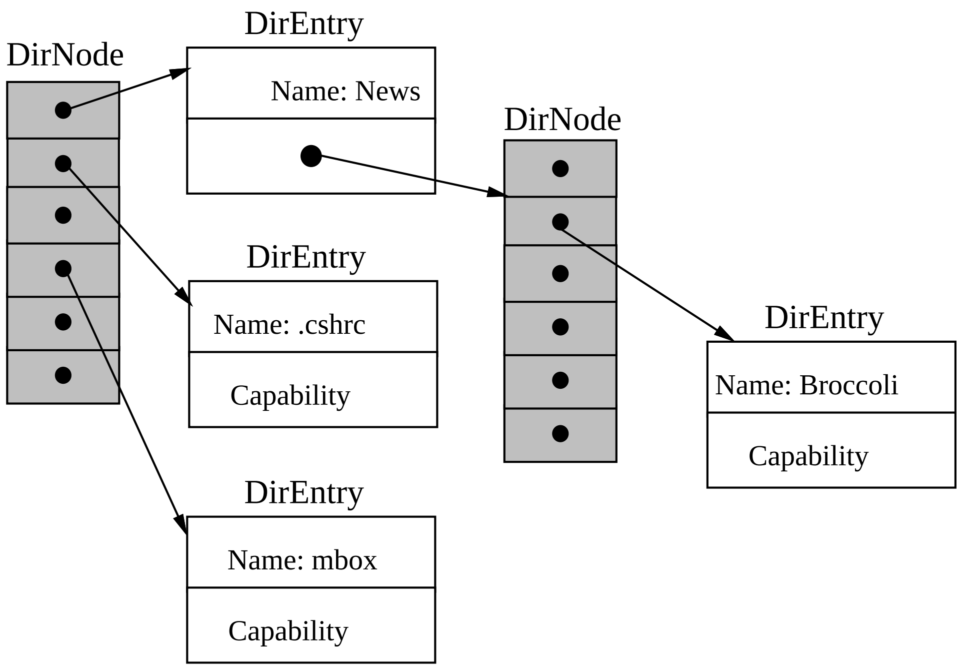

Figure 6.1: The internal organization of the name server

directory. Clients can resolve names to capabilities, browse directories, add new names and delete existing ones. The name server can, for example, be used as a simplistic object-based file server, where entries name capabilities for user files.

The organization of the internal data structure is shown in Figure 6.1. Two C++ classes are used to represent the structure, *DirEntry* and *DirNode*, which respectively models a name and a directory. A DirNode contains a single linked list of DirEntry objects, and a DirEntry stores either a capability or a pointer to a DirNode. Figure 6.2 shows the interface for a DirNode. Capabilities can be added, deleted and looked up by name with the methods *insert*, *remove*, and *resolve*. Similarly, directories can be manipulated with the methods *createDir*, *removeDir*, and *resolveDir*. The two functions *first* and *next* are provided for browsing through the entries in a DirNode. A pointer to the first DirEntry object is returned by the *first* method and the link from one DirEntry object to the next is returned by the *next* method. Figure 6.3 shows the interface for a DirEntry object.

Method			Read-only
`boolean`	`insert`	`(char *)`	
`boolean`	`remove`	`(char *)`	
`boolean`	`resolve`	`(char *, Capability&)`	$\surd$
`DirNode*`	`createDir`	`(char *)`	
`boolean`	`removeDir`	`(char *)`	
`DirNode*`	`resolveDir`	`(char *)`	$\surd$
`DirEntry*`	`first`	`()`	$\surd$
`DirEntry*`	`next`	`(DirEntry *)`	$\surd$

Figure 6.2: Interface for the DirNode Class

Method			Read-only
`DirEntry*`	`get`	`(char *, Capability&)`	$\surd$
`boolean`	`isSubDir`	`()`	$\surd$
`DirNode*`	`getSubDir`	`()`	$\surd$

Figure 6.3: Interface for the DirEntry Class

It provides methods for accessing the capability or the subdirectory depending on what the object contains.

Synchronization of concurrent access to the name server is done with locks for the write-methods and with version-based synchronization for read-only methods. The implementation is explained in detail in appendix A.

Shared proxies and read-only methods can with advantage be used to implement the client/server communication for the name server. Access to the name server is expected to consist mostly of browsing the structure and retrieving capabilities, i.e., mostly read-only access. Share proxies eliminate the overhead of creating local proxies in the client and reduce the time it takes to pass pointers from the server to the client. Read-only methods (see Figures 6.2 and 6.3) eliminate the expensive protected procedure calls, clients can follow pointers in the server's data structure directly.

6.2 Experiments

The benchmark consists of a client that creates, traverses, and deletes a directory structure in the server. The directory structure is a balanced tree with a branching level of two and a depth of ten. It contains a total of 3069 objects, where 2046 are DirEntry objects and 1023 are DirNode objects. The implementation of the operations is explained below.

Create

Creating the tree structure consists of executing a *createDir* method for all the internal nodes in the tree, and executing an *insert* for all leaf nodes. Only write-methods are executed.

Traverse

Traversing the tree structure is implemented as a recursive procedure. It uses *first* and *next* to iterate through the entries in a DirNode. For each entry it checks if it has a subdirectory with *isSubDir*, and whether it has one or not, a pointer to the subdirectory is fetched with a call to *getSubDir* and recursively traversed. Traversing the tree only requires invocation of read-only methods.

Delete

The tree is traversed recursively as explained above. After a node or leaf has been visit a call to *removeDir* or *remove* is generated, respectively. Deleting the structure involves executing a combination of read-only methods and write-methods.

To evaluate the effect of shared proxies and read-only methods four different implementations of the name server and client were built. The source code for the programs is unchanged for all implementations, but they are linked with different proxy and guard stub code. The implementations are:

Unprotected

The client and the server execute in the same protection domain without the use of *Porc* and stub code. The client invokes the server directly with ordinary procedure calls. This version gives a lower bound for the execution times.

Protected

The client and server execute in separate protection domains using the standard implementation of *Porc*. Calls to the server are done with RPC calls and neither shared proxies nor read-only methods are used.

Shared

The client and server execute in separate protection domains and shared proxies are used for all protected objects. Shared proxies are created lazily when a reference to an object is first exported to a client.

Read-only

In addition to using shared proxies, read-only methods are also used. The read-only methods return pointers to shared proxies, so the client can modify (or delete) the object that a pointer refers to. Shared proxies are created eagerly.

The experiments are run on a dedicated DECstation 5000 running Mach 3.0 and a prototype of the Opal[Chase et al. 93] server. The Opal server provides an environment in which all applications run in a single address space on top of the Mach kernel. *Porc* uses Mach ports as its transport mechanism, and read-only sharing of memory is directly implemented as *attach-segment* calls to the Opal server.

6.3 Results

The results of the experiments are shown in Table 6.1. The table shows the absolute times for coordinating sharing of memory (*attach*), creating the directory tree, traversing

Table 6.1: Performance Measurements (in seconds)

	Unprotected	**Protected**	**Shared**	**Read-only**
Attach	0.00	0.13	0.13	0.13
Create	0.29	0.77	0.75	0.80
Traverse	0.03	1.45	1.17	0.04
Delete	0.18	1.74	1.45	0.59
Total	0.50	4.09	3.50	1.52

it, deleting it, and the total execution time.

It comes as no surprise that the Unprotected version is much faster than the Protected version, because there is no overhead of executing remote procedure calls. However, inserting a protection boundary changes the relative cost of the different operations. For the Unprotected version allocating memory is an expensive operation compared to executing a procedure call, so creating the data structure is the most expensive task. For the Protected version, traversing and deleting the structure is more costly than creating the structure, due to the high cost of remote procedure calls. The Traverse and Delete operations make more than three times as many calls to the server than the Create operation.

Introducing shared proxies makes creating, traversing, and deleting the tree slightly faster. For Traverse the overhead of instantiating local proxies is eliminated. The performance is also improved because the server returns pointers directly to shared proxies, thereby eliminating hashtable[1] look-ups to convert capabilities into local proxy pointers. The results show that the extra cost of retrieving the Channel pointer from a thread's control block is dominated by the savings from passing proxy-pointers directly and eliminating instantiation of local proxies.

Attaching the name server's data segment read-only to the client significantly improves the performance of traversing and deleting the structure. Creating the structure is

[1]The hashtable used in the experiments has 2048 buckets.

slightly slower because shared proxies and guards are created eagerly. Traversing the structure in the Read-only version is a little slower compared to the Unprotected version, because the stub code for the read-only methods must convert pointers and call the method indirectly. The *first* and *next* methods only consist of returning a pointer stored in the server, so adding an extra level of indirection slows them down by a relatively large amount.

The total execution times show that the extra time spent attaching segments read-only to clients can easily be amortized by the use of shared proxies and read-only methods, thereby improving the overall performance of object-based RPC.

Chapter 7

Conclusion

An object-based RPC toolkit allows modular software systems to be split up in several protection boundaries to provide extensibility, maintainability, and independence of failure. It makes the protection boundary transparent to the client and provides protected access to the server. However, while the toolkit provides safety, it can suffer in performance, due to the domain-crossing costs required by the protected procedure calls.

This thesis focuses on how the performance of object-based RPC can be improved in the local case where both the client and the server reside on the same machine. In the local case, the client and the server can take advantage of read-only shared memory. Sharing of memory allows clients to directly query protected objects and makes it possible to share proxy objects between clients. Security is maintained, because protected procedure calls must still be used to modify protected objects. Three techniques are presented:

- Shared Proxies,
- Read-only Methods, and
- Version-based Synchronization

Shared proxies eliminate the cost of instantiating local proxies in clients, and allow servers to pass proxy pointers directly to clients. In addition they improve the transparency of the toolkit, by eliminating pointer aliasing and allowing sharing of proxy pointers between clients. Read-only methods provide clients with efficient read-only access to protected objects. The techniques have been evaluated on a test application where they show significant performance improvements. In particular, read-only methods shows a drastic performance improvements by eliminating the protected procedure calls.

Version-based synchronization, an efficient synchronization of concurrent access to protected objects with read-only methods, is presented. Version-based synchronization relies on detection of concurrent access and recovery. It provides fast access to server data in the common case where an object is not modified. The scheme introduces some problems with memory managements. A partial solution that recycles memory among objects of the same type has been presented.

The techniques have successfully been implemented in an existing object-based RPC toolkit (*Porc*). The modifications are completely transparent to existing applications, i.e., the same syntax and semantics have been preserved for all functions. *Porc* is built on top of Opal, a single address space operating system, which greatly simplified the implementation, because no a priori negotiation about sharing of memory between clients and servers is needed.

7.1 Future Work

There are several areas where future work is possible.

Real Applications

The techniques presented in this thesis are evaluated on a name server with a fairly artificial workload. Incorporating the methods into existing applications to gain more knowledge about their strengths and weaknesses would be interesting.

Distributed Virtual Memory Systems

The techniques could be used in the remote case on systems that implement distributed virtual memory[Chase et al. 92b, Heiser et al. 93, Carter et al. 91]. It is not clear how the techniques will perform compared to doing RPC calls directly to the remote host, and how they will interact with the underlying consistency policy of the DVM system.

Synchronization of Read-only Methods

Version-based synchronization has been presented as a way of efficiently synchronizing read-only methods and write-methods. The scheme needs further investigation to fully understand its viability.

Bibliography

[Accetta et al. 86] N. Accetta, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. MACH: A new kernel foundation for UNIX development. In *USENIX Summer Conference*, July 1986.

[Allchin & McKendry 83] J. Allchin and M. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 31–44, August 1983.

[Almes et al. 85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Anderson et al. 86] M. Anderson, R. D. Pose, and C. S. Wallace. The password-capability system. *The Computer Journal*, 29(1):1–8, February 1986.

[Bershad et al. 90] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[Birrell & Nelson 84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Carter et al. 91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164. ACM, October 1991.

[Chase et al. 92a] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-02, University of Washington, Department of Computer Science and Engineering, March 1992. Shortened version published as *Opal: A Single Address Space System for 64-Bit Architectures*, Third IEEE Workshop on Workstation Operating Systems (WWOS-III), April 1992.

[Chase et al. 92b] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.

[Chase et al. 92c] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1992. University of Washington CSE Technical Report 92-03-09.

[Chase et al. 93] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska. Sharing and protection in a single address space operating system. Technical Report 93-04-02, University of Washington, Department of Computer Science and Engineering, April 1993. To appear in ACM TOCS.

[Dave et al. 92] A. Dave, M. Sefika, and R. H. Campbell. Proxies, application interfaces, and distributed systems. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, September 1992.

[Feeley et al. 93] M. J. Feeley, J. S. Chase, and E. D. Lazowska. User-level threads and interprocess communication. Technical Report 93-02-03, University of Washington, Department of Computer Science and Engineering, March 1993.

[Heiser et al. 93] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: A distributed single address-space operating system. Technical Report SC&E Report 9314, School of Computer Science and Engineering, The University of New South Wales, 1993.

[Herlihy 91] M. Herlihy. A methodology for implementing highly concurrent data objects. Technical Report CRL 91/10, DEC Cambridge Research Laboratory, 1991.

[Hoare 78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Jul et al. 88] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Levy 84] H. M. Levy. *Capability-Based Computer Systems.* Digital Press, Bedford, Massachusetts, 1984.

[Mullender & Tanenbaum 86] S. Mullender and A. Tanenbaum. The design of a capability-based operating system. *The Computer Journal*, 29(4):289–299, 1986.

[Redell et al. 80] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Rosenberg 92] J. Rosenberg. Architectural and operating system support for orthogonal persistence. *Computing Systems*, 5(3), July 1992.

[Shapiro 86] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the Sixth International Conference on Distributed Computing Systems*, May 1986.

[Stroustrup 91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[Wulf 74] W. A. Wulf. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

Appendix A

Version-based Synchronization: An Example

This appendix gives an example of how version-based synchronization can be implemented. A singled linked list that provides read-only methods to access and iterate through the list is used as an example. The implementation described here is identical to the one used for implementing the *DirNode* class in the name server[1].

The list has methods for inserting elements, removing elements, and iterating through the elements in the list. For simplicity are the type of the elements stored in the list integers in this example. The list is implemented with two C++ classes, whose declarations are shown in Figure A.1. Each integer stored in the list is encapsulated in a *ListElem* object, which also contains the link-pointer to the next element. The *List* class contains a pointer to the first element in the list and has access to the internals of ListElem, so it can access the link-pointer from one element to the next. Both of the classes are subclasses of *LockFreeObject* (Figure A.2) which contains the version-numbers and provides methods to manipulate them.

[1]The name server is described in Section 6.1

```
class List : public LockFreeObject {
   Mutex lock;
   ListElem *head;
public:
   void insert(int key);
   int  remove(int key);

   ListElem *first ()              // read-only
   ListElem *next  (ListElem *)    // read-only

   void *operator new(size_t);
   void  operator delete(void *);
};

class ListElem: public LockFreeObject {
   int key;
   ListElem *link;

   friend class List;
public:
   int  getKey();                    // read-only

   void *operator new(size_t);
   void  operator delete(void *);
};
```

Figure A.1: The List and ListElem Classes

The following sections describe how memory is managed for the list, and explains how the read-only methods and the write-methods are implemented.

A.1 Memory Management

Recycling of memory is implemented by overloading the `new` and `delete` operators. An instance of the *LockFreeMemPool* class (see Figure A.3) contains the free-list of objects. The LockFreeMemPool is a simple queue of LockFreeObject elements, where objects can

```
class LockFreeObject {
   ulong        check[2];
   LockFreeObject *link;         // Used by LockFreeMemPool
   friend class LockFreeMemPool;
public:
   ulong lf_first()           { return check[1]; };
   ulong lf_last()            { return check[0]; };

   void  lf_inconsistent()    { check[0] = check[1]+1; };
   void  lf_consistent()      { check[1]++; };
};
```

Figure A.2: The LockFreeObject Class

be pushed onto the queue with the *recycle* method, and popped off with the *allocate* method. The *allocate* method returns a null pointer if the queue is empty. The class has access to the internals of LockFreeObject and uses its *link* pointer to queue elements.

There is a global instance of the LockFreeMemPool class for each type of object that is accessed with read-only methods, i.e., types that are a subclass of LockFreeObject. For the List and ListElem class, the following statements instantiate the two queues:

```
static LockFreeMemPool ListMemPool;
static LockFreeMemPool ListElemMemPool;
```

The code for the overloaded `new` and `delete` operations are the same for all classes, except that each class accesses its own LockFreeMemPool and casts to its own type. The code for the List class is shown below.

```
void *List::operator new(size_t s) {
   List *elm = ListMemPool.allocate();

   return (elm) ? (elm) : (::new List);
}

void operator List::delete(void *obj) {
```

```
class LockFreeMemPool {
   Mutex *mutex;                // Mutex
   LockFreeObject *head,*tail;

   public:
      LockFreeMemPool();
      ~LockFreeMemPool();

      LockFreeObject *allocate ();
      void           recycle  (LockFreeObject *obj);
};
```

Figure A.3: The LockFreeMemPool class

```
      ListMemPool.recycle(obj);
   }
```

A.2 Read-only Methods

Read-only methods are executed directly by clients, and therefore cannot modify any state of the server. Version-based synchronization of read-only methods relies on detecting and recovering from reading inconsistent data. In contrast to locks which are based on prevention. The technique relies on write-methods to mark objects as inconsistent while they are being modified.

The guidelines for implementing read-only functions are:

- No code modifies the object,
- All access to the object is enclosed in a do-while loop, so the method can be re-tried if the version-numbers changed during its execution, and
- Only local variables are accessed outside the do-while loop

The implementation of the read-only methods in the List and ListElem is shown in Figure A.4. All of the methods have the same structure. The only difference is what data is accessed and which object's version-numbers are checked.

A.3 Write Methods

Write-methods are executed by the server, so they can freely modify, create, and delete objects. In particular, they can synchronize access to shared data with standard synchronization primitives such as critical sections and monitors[Hoare 78]. A write-method must modify the version-numbers before and after it modifies an object to prevent read-only methods from returning inconsistent data. The implementor does not have to access the version-numbers directly, but can manipulate them via the methods in the LockFreeObject class.

The implementation of the *insert* and *remove* methods of the List class is shown in Figure A.6. A lock (the *mutex*) is used to ensure that only one thread is inserting or removing elements at a time. Before changes are applied to an object it is marked inconsistent with a call to *lf_inconsistent*, and afterwards it is marked consistent with a call to *lf_consistent*. These calls guarantee that concurrent executing read-only methods will be restarted.

One subtlety with the implementation is the interaction between read-only methods and write-methods. What happens if elements are added and removed while a client is traversing the list with the read-only *first* and *next* methods?

Consider the case where only elements are added to the list. New elements are always put at the beginning of the list. Elements that are added after the client obtains a pointer to the first element (with the *first* method) will therefore not be traversed. Removing elements from the list is a little different. Removing an element that the client has already traversed will make no difference, because it has already been accessed. Similarly, the

```
int ListElem::getKey() {
   ulong first, last;
   int  k;

   do {
      first = lf_first();
      k=key;
      last  = lf_last();
   } while(first!=last);

   return k;
}

ListElem *List::first() {
   ulong first, last;
   ListElem *h;

   do {
      first = lf_first();
      h = head;
      last  = lf_last();
   } while(first!=last);

   return h;
}

ListElem *List::next(ListElem *e) {
   ulong first, last;
   ListElem *n;

   do {
      first = e->lf_first();
      n = e->link;
      last = e->lf_last();
   } while(first!=last);

   return n;
}
```

Figure A.4: Implementation of the read-only methods

```
int ListElem::getKey(ulong i) {
   ulong first, last, I;
   int  k;

   do {
      first = lf_first();
      k=key;
      I= lf_incarnation();
      last  = lf_last();
   } while(first!=last);

   if (i!=I)
      //Raise exception
   else
      return k;
}
```

Figure A.5: Implementation of getKey with incarnation check

client will not see objects that are removed from the part of the list that has not yet been traversed. The element that the client is currently holding a pointer to is guaranteed to be traversed, even if it is deleted while the client is accessing it. When the element is deleted it is just being put on a free-list with the *recycle* call to a LockFreeMemPool object with all its state intact.

However, because memory is recycled, the implementation of the linked list has an abnormality. It occurs when the current accessed element is removed from the list and its memory region is recycled to a new element. The new element is inserted into the front of the list, so the traversal will, in effect, be reset to the beginning of the list, making it possible for a client to traverse some elements more than once. This behavior might be undesirable for some applications. For example, a client cannot traverse the list to determine its length. An even worse situation can happen if the element is deleted and recycled onto a different list.

An alternative implementation strategy uses incarnation numbers. This makes it possible for a read-only method to detect when the object it is accessing has been recycled, so an error code can be returned to the application. The application can then restart its operation by retrieving a new pointer to the start of the list.

The idea is to let the clients keep track of which incarnation of an object a pointer refers to. The incarnation number is an instance variable in the LockFreeObject class[2]. It is incremented each time an object is recycled, hence, it is constant as long as the same memory region contains the same object. Instead of only returning the pointer to an object, the server returns a two-tuple contain the pointer and the incarnation-number. The incarnation number is passed to the server when the pointer is used, so the object can be authenticated. An example of the technique is shown in Figure A.5. Unfortunately, implementing read-only methods with incarnation-numbers are not completely transparent to clients, which jeopardize the goals of the toolkit.

[2]It is not shown in Figure A.2

```
void List::insert(int key) {
    mutex.lock();
    ListElem *elm = new Listelem;

    elm->lf_inconsistent();
    elm->key  = key;
    elm->link = head;
    elm->lf_consistent();

    lf_inconsistent();
    head = elm;
    lf_consistent();
    mutex.unlock();
}

void List::remove(int key) {
   NameServerEntry_r *cur = head,
                     *prv = NULL;

   mutex.lock();

   while(cur && key!=cur->key);        // search
      prv=cur,cur = cur->link;

   if (cur)) {
      if (prv) {                       // remove
         prv->lf_inconsistent();
         prv->link = cur->link;
         prv->lf_consistent();
      } else {
         lf_inconsistent();
         head = cur->link;
         lf_consistent();
      }
      delete cur;
   }
   mutex.unlock();
}
```

Figure A.6: Implementation of the write-methods