Lightweight Shared Objects in a 64-Bit Operating System

Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Abstract

Object-oriented models are a popular basis for supporting uniform sharing of data and services in operating systems, distributed programming systems, and database systems. We term systems that use objects for these purposes *object sharing systems.*

Operating systems in common use have nonuniform addressing models, making the uniform object naming required by object sharing systems expensive and difficult to implement. We argue that emerging 64-bit architectures make it practical to support uniform naming at the virtual addressing level, eliminating a key implementation problem for object sharing systems.

We describe facilities for object-based sharing of persistent data and services in Opal, an operating system we are developing for paged 64-bit architectures. The distinctive feature of Opal is that object sharing is supported in a runtime library, above a single virtual address space that maps all primary and secondary storage in a local area network.

This work was supported in part by the National Science Foundation under Grants No. CCR-8619663 and CCR-8907666; by the Washington Technology Center; and by Digital Equipment Corporation through the Systems Research Center, DECwest Engineering, the External Research Program, and the Graduate Engineering Education Program.

1 Introduction

The purpose of an operating system is to allow untrusting users to develop and execute applications. These applications must be able to store and retrieve information, protect that information, and share that information with other applications in a distributed environment. An operating system must be extensible, so it must allow new interfaces, services, and languages to be implemented *as applications*, without affecting the underlying system.

The concept of abstract objects existing in a uniform object name space has long been used to achieve these goals. Because objects are *self-describing* (their methods are dynamically bound), applications can operate on objects encountered in shared or persistent storage. Because objects are *encapsulated*, they can represent protected services whose use is restricted; a client must possess an unforgeable object reference (a *capability*) for the service, and can operate on the service only by invoking its methods. Because objects are *abstract*, object method invocation is syntactically and semantically independent of the object's location in the system. These object properties make object-oriented languages an attractive basis for language-integrated and type-checked access to shared and persistent data and services.

These ideas have been used in object-oriented database systems, persistent object stores, and distributed programming systems [Jul et al. 88, Bal & Tanenbaum 88]. Most microkernel operating systems (e.g., Mach [Young et al. 87], Amoeba [Mullender & Tanenbaum 86], and Chorus [Rozier et al. 88]) represent protected system services as objects named by capabilities that

OOPSLA'92, pp. 397-413

can be passed between applications. These different uses of objects have been combined in operating systems that define all interactions between system components in terms of objects [Marques & Guedes 89, Shapiro et al. 89]. The common theme in all of these efforts is the use of object models to define and support sharing of data and services between applications. We refer to these systems generically as *object sharing systems.*

1.1 The Problem of Object Identity

Designers of object sharing systems face difficult choices regarding which level of the system should support object sharing semantics. Object sharing may be supported by runtime libraries, servers, programming languages, the operating system kernel, and/or the hardware. The choice of levels determines the answers to critical questions about the system. What is the cost of object support? Are users who do not want the support forced to pay for it? Are multiple object models supported? Can new models be defined?

The choice of levels is driven by the key property of objects important for object sharing systems – the uniform object name space extending across the scope of the sharing. This property of *object identity* [Khoshafian & Copeland 86] allows object names (pointers or capabilities) to retain their meaning when they are stored in databases and passed between applications. In most object sharing systems today, object identity is implemented by translating object names in software or special-purpose hardware as data moves around the system; this translation is complex and expensive, and it requires knowledge of where names are stored and how they are used. Systems always choose the cheapest naming solution that meets their sharing needs:

- Most programming language implementations - with no support for sharing - name objects with virtual addresses, avoiding the expense of software pointer translation.
- Object-oriented database systems use *surrogate pointers* translated to virtual addresses by the runtime system. Surrogates are sometimes called *soft pointers* or *persistent pointers.*
- Schemes for distributed objects and protected services may involve the operating system kernel, since the name binding must be unique across network or protection boundaries.

This leads to a proliferation of incompatible shared object implementations of varying expense and complexity.

1.2 Large Address Spaces

We believe that the impending arrival of machines with large virtual address spaces will allow us to unify these different styles of shared objects in a simple and efficient way. The DEC Alpha [Dig 92] and the MIPS R4000 [MIP 91] are recent examples of wide-address architectures.

Wide-address architectures remove the basic restrictions underlying the problem of nonuniform naming in computer systems. Existing operating systems have impeded the development of object sharing systems – in fact, they do not support sharing well in any form, because their basic addressing models are nonuniform; every application executes in a private virtual address space, and long-term storage is placed outside of these address spaces. This nonuniformity is both the primary motivation for object-based approaches to sharing and the primary implementation problem for object sharing systems. The cause of this nonuniformity is the small address spaces of today's architectures, on which virtual addresses are a scarce resource that must be multiply allocated and are therefore ambiguous. We believe that the need for this ambiguity will soon disappear. A full 64-bit address space can map 16 billion gigabytes of data; if consumed at the rate of 100 megabytes per second this address space would last for nearly 5000 years.

We are building a new operating system called Opal [Chase et al. 92a, Chase et al. 92b] that exploits the wider virtual addresses of these emerging architectures by defining a *single virtual address space* that includes data on long-term storage and across a local area network. Hardware-based memory protection exists within this uniform address space; programs execute in *protection domains* that restrict their access to global virtual storage. Our goal in this paper is not to explain or justify this system (we refer you to [Chase et al. 92a] for a discussion of the system and its relationship to previous work). Nor is our purpose to argue for 64-bit hardware; we assume that wide-address hardware will become widely available. The purpose of this

paper is to explore the effect of Opal's uniform virtual address space on object sharing systems.

1.3 Sharing in a Global Address Space

In this paper we argue that a uniform virtual address space is the right level of operating system kernel support for object sharing systems. Our basic thesis is that object sharing, object persistence, and capability-based service protection can and should be implemented *outside* of the operating system kernel, without name translation, and that this can be done simply and efficiently when the operating system provides appropriate virtual memory support, particularly a uniform virtual address space.

To illustrate these points, we describe a runtime support package layered above Opal's basic system abstractions. This package supports shared and persistent data objects, and *protected objects* that represent system services isolated from their clients by memory protection boundaries. Our discussion focuses on *system* issues relating to naming, binding, and protection of objects, sidestepping *language* issues such as the type model or the programmer's view of concurrency. The goal is to understand the opportunities of a 64-bit address space and to raise new questions about the relationship between shared objects and virtual memory.

The paper is organized as follows. In the next section we outline the basic Opal abstractions. Section 3 describes our object model and presents our views on the relationship between language-level objects and the underlying kernel and hardware abstractions. Section 4 explains how Opal's virtual memory model allows data objects to be persistent and/or passed/shared between applications, and outlines some assumptions and tradeoffs of our approach. In Section 5 we describe the use of objects to represent protected services, focusing on the relationship between hardware-protected service objects and language-protected data objects. Section 6 presents our conclusions.

2 The Opal Operating System

In this section we outline the basic Opal abstractions at a level of detail sufficient to understand the object sharing issues discussed in the rest of the paper. As previously stated, Opal defines a single virtual address space that maps all data in the system, including persistent data. This simply means that virtual address usage is coordinated so that shared data is addressed with the *same names* by all programs that access it. Memory protection is independent of this global name space. No special hardware support (other than wide virtual addresses) is needed for our approach.

Opal is designed for a distributed environment composed of one or more *nodes* connected by a message network. Each node contains physical memory, one or more processors attached to that memory, and possibly some long-term storage ("disk"). We assume that the processors are homogeneous. The global address space is extended across the network by placing servers on each node that maintain a partitioning of the address space, both to ensure global uniqueness and to allow data to be located from its virtual address. The problems are similar to those faced by other systems for distributed name management.

2.1 Opal System Model

Listed below are the four basic facilities that operating systems provide, and the Opal abstractions that represent them. Opal's abstractions are similar to the abstractions of other microkernel operating systems, with some crucial differences that will become clear later. The central point of this paper is that all object support should be built above these basic abstractions; no explicit kernel support for shared objects is necessary or desirable.

- **Execution.** A *thread* is an executing stream of instructions and its procedure call stack. An executing application may contain multiple threads. Threads are multiplexed on processors, and different threads may execute in parallel on different processors. A thread can be suspended (e.g., for blocking synchronization or timeslicing) and resumed later, possibly on another processor. Threads are implemented in a runtime library using techniques described in [Anderson et al. 91].

- **Data Storage.** All data (including executable code) is stored in *virtual segments*, sequences of virtual pages occupying a contiguous range of virtual address space (Figure 1). We say that segments are "virtual" because

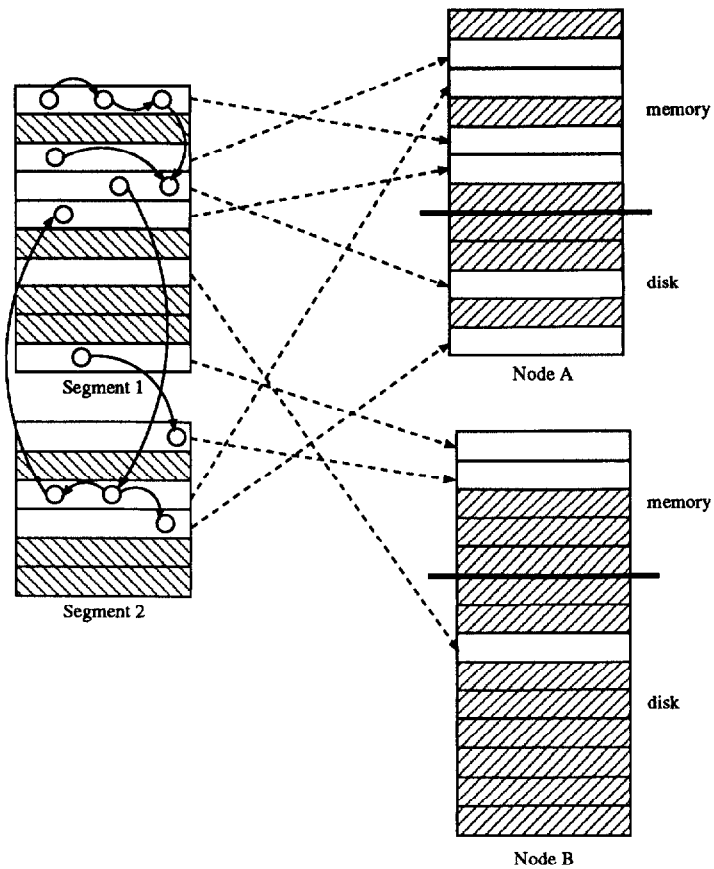

Figure 1: Virtual and Physical Storage

the operating system transparently manages some aspects of the physical storage backing them: (1) when data is first written into a virtual page, the system allocates a physical page to back it, (2) the system may move a physical page between node memories and disk, or it may make multiple copies of the page, (3) when a thread references a virtual page a copy of the page is faulted into the memory of the thread's node.

- **Protection.** Untrusting applications must be prevented from damaging each other's internal data. Threads execute within *protection domains* that limit their access to storage; typically, all of the threads of an executing application run within the same domain. A thread cannot complete a reference to a virtual page unless the segment containing that page is *attached* to the thread's domain (Figure 2). In general, each domain has permission to attach only a small subset of segments of the global virtual store, with some combination of **read**, **write**, and **execute** privilege.

- **Communication.** Domains can communicate implicitly by reading and modifying data in shared segments. In addition, control can be transferred between domains through *portals*. A portal is an entry point to a domain, uniquely identified by a 64-bit value. Any thread that knows the value of a portal identifier can make a kernel call that transfers control into the portal's domain. Conceptually,

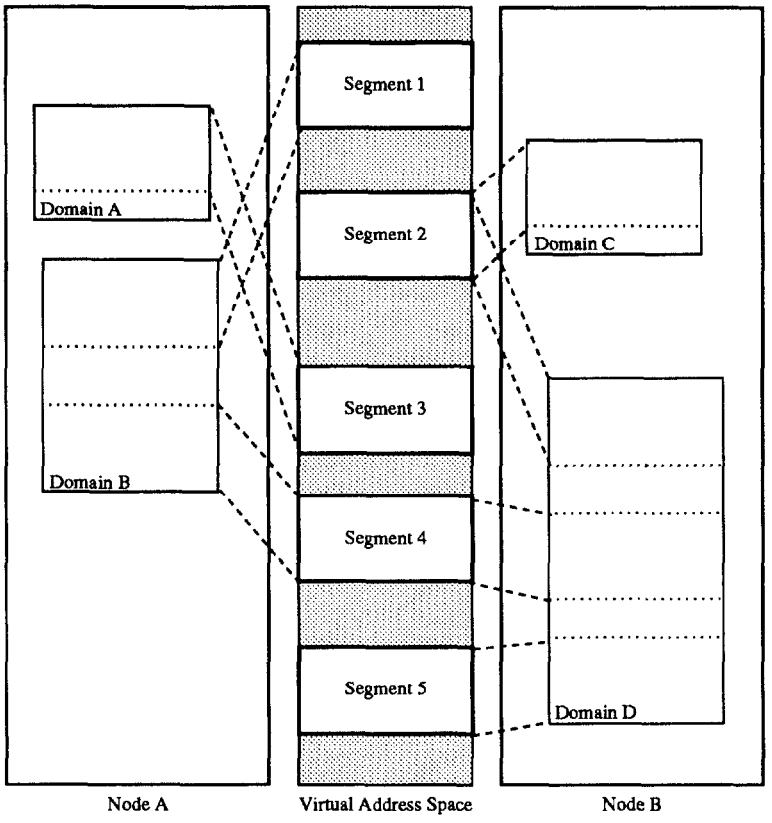

Figure 2: Protection Domains and Virtual Segments

the calling thread switches into the portal's domain and begins executing at a virtual address chosen by the owner of the portal.

Portals are the basis for a *cross-domain procedure call* abstraction (also called *remote procedure call* or RPC). The model for an RPC call is that a thread in a *client* domain switches to a *server* domain to execute a protected procedure call, then returns to the client domain when the operation completes. This can be thought of as passing a message and receiving a response; in fact, the implementation uses messages if the client and server are on different nodes. If the client and server are colocated, arguments and results can be passed through a small segment called a *channel* that is shared by the client and the server (this technique for *lightweight* RPC is described in [Bershad 90]).

2.2 Segments and Addressing

In most operating systems, including segmented systems such as Multics [Daley & Dennis 68], each protection domain has a private virtual address space; a segment may be mapped to a different virtual address range by each domain that attaches it. In contrast, Opal has a single virtual address space shared by all protection domains. Each Opal segment occupies a fixed range of virtual addresses, assigned when the segment is created, and disjoint from the address ranges occupied by all other segments. Domains that attach to a given segment name it with the same virtual addresses. This means that linked pointer structures within and across segments are always meaningful to any domain that attaches them.

Segments appear to clients as protected objects named by capabilities (see Section 5) that support the operations **Attach** and **Detach**. However, it

is misleading to think of a segment itself as an object; threads in a domain can access an attached segment with **load** and **store** instructions, violating object encapsulation. A segment is a container for raw data; a *segment object* is a control block representing that container.

There are two ways that domains can attach to segments. First, domains can pass capabilities for segment objects and attach them explicitly. Second, the system can transparently locate and attach a segment in response to a *segment fault* generated by a reference to an unattached segment. In this case access control lists are used to determine permission.

Opal segments subsume *files* in conventional systems. Like Unix files, segments have active and passive reference counts that determine when they are reclaimed. If a segment has a nonzero passive count (e.g., there is a symbolic name for it in the name server) then the segment persists across system restarts.

2.3 Managing the Global Address Space

Our description leaves many aspects of the system model unspecified. How is access to segments determined? How are physical pages managed so as to provide a coherent view of global virtual storage, and recoverability after a system crash? When can physical storage be deallocated? Who manages segment reference counts? The answers to these questions are *policies* chosen on a segment basis according to the desired semantics and how much the user is willing to pay for them. We will return briefly to these questions in Section 4.3, where we discuss the role of objects in implementing these policies. For the moment we ask that the reader suspend disbelief and assume that the system can transparently manage storage in this distributed and persistent virtual address space.

3 Objects In Opal

We now turn our attention to the nature of object support in Opal. But first we point out that an object-oriented data model is not *necessary* to support sharing above a global virtual store. One goal of an object sharing system is to allow data structures (composed of objects linked by pointers) to be passed between applications, or stored on disk and transparently retrieved on pointer dereference. In principle, Opal's global virtual address space supports these properties by default, independent of object-orientation. A major reason for the current popularity of object-oriented approaches to sharing is that they create well-formed pointer graphs whose structure can be determined with help from the type system, and are therefore amenable to pointer translation. These constraints are no longer necessary in a uniform virtual address space.

However, we believe that object-oriented data models continue to be valuable for organizing and controlling sharing. Domains sharing data must have a common understanding of its internal structure and usage, best formalized as a common set of types. The purpose of Opal's object support package is to allow the code that implements object methods to be located and bound dynamically given an object reference, and to ensure that all domains operating on the data use the same implementations of those types. Dynamic binding also allows an application to use objects created by a different application and passed to it in shared storage, and new versions of a type can be introduced without affecting existing stored instances of old versions of the type. In addition, objects are a convenient granularity for distributed coherency, stable storage updates, and garbage collection.

In this section we describe our object model and the runtime representation of objects. The basic model is of passive fine-grained objects invoked by synchronous procedure calls from concurrent threads of control. The object representation is matched to a restricted form of the C++ programming language. This should be viewed as a "hypothetical" object sharing system; the restrictions are merely to simplify our exposition and prototyping effort. The premise is that a more comfortable language model would not significantly affect the underlying system issues.

3.1 Objects and Types

Our terminology and object representations follow [Shapiro et al. 89]. In this paper we use the term *object* to mean an *elementary object* that contains no internal linked data structures. This is a definition rather than a restriction. An object may contain pointers to other objects, so elementary objects can be linked together to form *composite ob-*

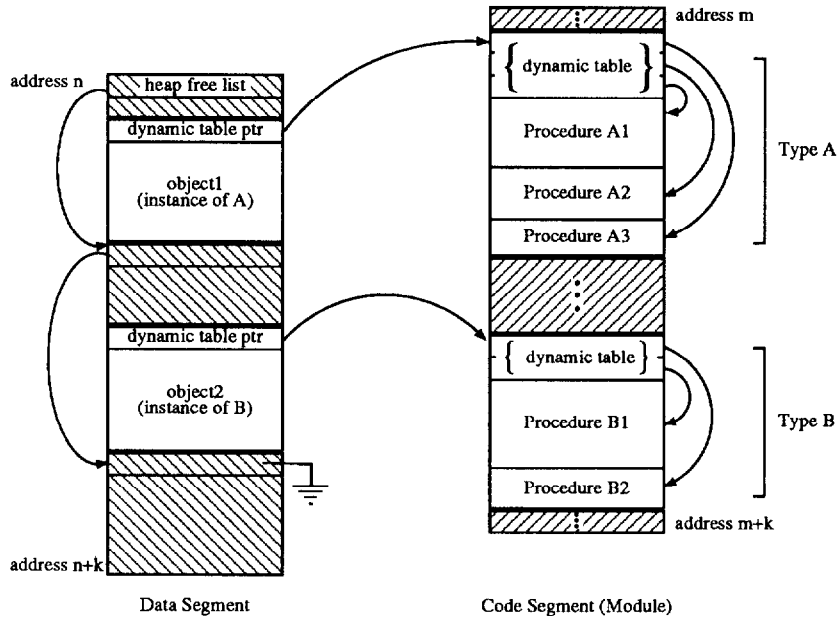

Figure 3: Objects and Types

jects of arbitrary complexity.

For our purposes, an object is a contiguous sequence of bytes that is an instance of a language type. [1] Its size and internal structure are statically determined by its type, and known only to the type implementation. An object reference is the virtual address of the first byte of the object.

The runtime representation of a type is a group of method procedures, and a vector of pointers to those procedures called the *dynamic table*. Types reside in segments called *modules* that can be attached readonly to any domain that trusts them. The first field of each object is a pointer to the dynamic table for the object's type (Figure 3). Callers of an object method are assumed to have static knowledge of the desired procedure index and call signature. This assumes a simple type system with no inheritance. A richer type system might result in a more complex object representation, but this is not important.

[1] We will occasionally refer to elementary objects as *language objects* or *data objects*.

3.2 Segments Are Containers

A *data segment* is a virtual segment with a self-contained heap. Objects are allocated from a heap with calls to a runtime library (C++ **new** and **delete**). Each elementary object is entirely contained within a data segment, though composite objects can span data segments. To allocate or free an object, a thread must be executing in a domain that has the object's data segment write-attached.

When a thread allocates an object it must specify which data segment to use. Opal's thread package associates a default heap for object allocations with each thread; multiple threads can share a heap, and the default heap is inherited across thread creation. A thread's default heap can be changed with a call to the runtime system.

Our use of data segments as heaps for allocating objects makes them similar to *clusters* in Oisin [Cahill 88]. In contrast, the original COOL system [Abrossimov et al. 90] viewed objects as composed of one or more segments; the COOL designers have pointed out that COOL objects are too expensive to represent most language objects. From COOL we learn that a general-purpose object sharing system must support small objects cheaply.

3.3 Concurrency and Synchronization

Concurrency is formalized as active threads synchronously invoking methods of passive objects. If an object is invoked by multiple threads then there is logical concurrency within the object, and possibly physical concurrency if the node has multiple processors. This concurrency is inherent in our passive object model. We assume that all type implementations are reentrant and thread-safe.

Synchronization is managed with explicit locking primitives. Threads and synchronization are implemented in a concurrency library similar to [Faust & Levy 90], providing mutexes, condition variables, and spinlock types. Threads and locks are objects allocated from data segment heaps. Lock objects contain flag words that are set and cleared with atomic instructions (e.g., **test-and-set**). Thread objects include a runtime stack, control information, and register state. This register state is loaded into a processor when the thread is scheduled to run, and written back to memory when the thread blocks.

4 Shared and Persistent Objects

As previously stated, the properties of object sharing and persistence derive directly from Opal's shared and persistent virtual address space. Objects are contained within segments, and segments can be shared between domains and made persistent, as described in Section 2. The single address space ensures object identity for objects in shared and persistent segments, because the virtual address at which an object appears is independent of (1) the location of the physical page, and (2) the protection domain referencing it. The dynamic table pointers have a globally unique interpretation for the same reason. Transparent dereference and dynamic code loading for shared and persistent objects is driven by ordinary page faults. Caching of persistent data is managed by the virtual memory system, avoiding two-level caching problems.

Synchronization is independent of sharing and persistence. Like all objects, lock objects can be stored on disk and retrieved, and invoked by threads from different domains. User-level (rather than kernel-level) support for threads and synchronization is essential for this property. This means that sharing and persistence are transparent to properly synchronized types, and most type implementations can be written without concern for how instances of the type are shared.

4.1 Shared Objects in Virtual Storage

Our proposal supports shared data objects above the virtual memory system by clustering objects into segments, and allowing domains to attach segments and directly address the objects they contain. There are several tradeoffs inherent in this approach. We view these tradeoffs as essential for high-performance object sharing on page-based architectures. Indeed, these tradeoffs are the defining characteristics of object-oriented database systems.

- *Object encapsulation for shared data is enforced by the type system rather than by hardware.* In principle, any domain that is given permission to write-attach a segment is free to misuse its objects, violate synchronization conventions, or even corrupt the heap management structures. Once access to a segment has been granted, the customer relies on strong typing in the programming language to prevent damage. The performance win is that language protection is much cheaper than hardware-based memory protection.

- *Access control is based on segments, not objects.* The system can restrict attach-access to an entire segment but not to individual objects within a segment. Object-grain access control can be bought by placing each object in a private segment, but this is wasteful of memory for small objects because the minimum segment size is a page. Segment-grain access control is not a significant restriction because data objects are usually used in clumps; applications operate on a graph structure rather than individual objects within that structure.

- *Applications must manage the assignment of objects to segments.* If an application is using more than one data segment, it must choose which segment to allocate each new object from. The choice must be made carefully; if two objects reside in the same segment it is impossible to grant attach-access to one but not the other.

4.2 Object Referencing With Virtual Addresses

The key point is that a uniform virtual address space permits the use of unencoded virtual addresses as object references in shared and persistent data. This is because each object resides at a unique and fixed virtual address, even if it is not in active use by any application. In contrast, virtual address bindings for shared objects are unstable in conventional operating systems, for two reasons: (1) each protection domain has a separate virtual address space, and (2) virtual addresses for objects in persistent storage are bound late. For this reason, today's persistent object systems use *surrogate pointers* rather than virtual addresses as object references.

Surrogate pointers require special support from the compiler in order to insulate the application from the need to manage two pointer forms. Also, software must use mapping tables to translate surrogates into virtual addresses before dereferencing them. This translation is expensive, even in systems that carefully localize and cache the mapping tables [Moss 90]. *Swizzling* is a technique to amortize this cost over multiple dereferences of the same pointer, by overwriting the surrogate in place with the virtual address after the first translation. The main drawbacks of swizzling are:

- It adds to the overhead of pointer translation because it requires copying the data as well as translating the pointers. Swizzled pointers must be translated in both directions, on output as well as input.
- It interferes with virtual memory caching because it dirties pages that are never modified by the application.
- It inhibits sharing because each domain must maintain a separate copy of the data, since the pointers are swizzled to different virtual addresses in each protection domain.
- It needs help from the compiler to locate pointers stored in the data.

Furthermore, systems that use swizzling must choose between "eager" and "lazy" variants, each of which has additional disadvantages. *Eager swizzling* (e.g., [Wilson 91]) translates all pointers as data is read into memory; it further increases the overhead of translation because it unnecessarily translates pointers that are never used by the application. Faster processors cannot reduce this cost proportionately, because eager swizzling is memory-intensive rather than CPU-intensive. In contrast, *lazy swizzling* translates pointers on first use; it requires additional support from the language implementation to trap dereferences of unswizzled pointers, and the translation latency cannot be overlapped with I/O.

Surrogates and swizzling are necessary compromises for narrow-address machines, but they should not be viewed as satisfactory solutions to the problem of uniform object referencing. Virtual address pointers are simpler and more efficient, and they can be used directly as object references given wide-address hardware and proper operating system support for a uniform virtual address space. [Copeland et al. 90] outlines a similar proposal, and presents a case for the use of virtual addresses as persistent object references.

One disadvantage of virtual address pointers is that moving an object to a different part of the address space invalidates all references to it. Objects may need to be moved to compact storage or improve clustering properties, particularly since the address space contains persistent data. This need can be met (when it exists) by (1) using one or more levels of pointer indirection for object addressing, or (2) leaving forwarding records for objects that have moved, but both of these solutions introduce new problems and add to the cost of pointer dereference. Alternatively, virtual page protections can be used to trap references to old copies of objects that have moved [Appel et al. 88].

4.3 Summary

In this section we have described support for shared and persistent data objects in Opal. The highlights of our approach are: (1) clustering of data objects within segments, (2) use of segment attachment and direct virtual memory access as the basic mechanism for object sharing, and (3) object naming and code binding using unencoded virtual addresses. However, several aspects of sharing and persistence have not been addressed in this discussion. These are active research areas, in which knowledge of the partitioning of data into objects can be exploited by the system.

- *Garbage collection.* We recognize that garbage collection is essential for a fully general object sharing system. We rely on explicit deletion at all levels in our prototype, but this is error-prone and does not work for unpredictable sharing patterns. Opal supports multiple protected reference counts on segments, for direct use by applications or by a garbage collector integrated with the programming language (and presumably running in a privileged domain that attaches all segments from a particular language environment). We claim that garbage collection is no more difficult than ever if appropriate programming languages are used, that the system can use accounting to encourage reclamation just as conventional operating systems do, and that the protection mechanisms prevent untrusting domains from harming each other with reclamation errors. These are untested hypotheses.

- *Consistency of distributed and persistent segments.* Several approaches to the problems of consistency and recoverability (transaction support) have been developed. There are pure page-based solutions (e.g., Ivy [Li 86] and Camelot [Eppinger 89]), pure object-based solutions (e.g., Amber [Chase et al. 89, Feeley & Levy 92] and Argus [Liskov et al. 87]), and interesting hybrid solutions (e.g., Munin [Carter et al. 91] and Avalon [Detlefs et al. 88]). Our initial prototype uses primitive mechanisms, but any of these approaches could be used to ensure consistency and/or recoverability of an Opal segment.

These are existing problems that we have failed to resolve rather than new problems that we have introduced. Our basic approach to these issues is to build a framework that is flexible enough to accommodate the various solutions that have been developed, and to allow multiple solutions to coexist. That is, we believe that the question of how storage is *managed* is largely independent of the way that storage is *named*; the latter is the primary focus of this project.

5 Protected Service Objects

Until now we have mostly been concerned with supporting shared and persistent *data*. Objects can also be used to represent protected *services* whose clients are not trusted to attach them directly. In this section we review the use of object models for protected services, and describe support for object-oriented servers in Opal.

The key object property for services is *encapsulation*, which has two aspects:

- *Enforced Integrity.* A client cannot maliciously or accidentally corrupt the service because it cannot operate on it except by invoking its methods. The method code is chosen by the implementor of the service rather than by the client.

- *Object-grain access control.* A client cannot invoke a service unless it holds a reference for the object that implements the service. A client cannot hold a reference unless another entity with access to the service passed it a reference. Access checking occurs when the reference is first passed to the client, but is unnecessary on subsequent calls.

Programming languages enforce object encapsulation through the type system. In system environments, where unsafe programming languages may be used, encapsulation is enforced by two means: (1) the object is separated from the client by a hardware-enforced memory protection boundary, and (2) the system or the hardware supports special object references called *capabilities* [Fabry 74] that cannot be forged; the client must hold a capability for the object in order to invoke it with a cross-domain (RPC) call. An object that is encapsulated in this way is called a *protected object* or a *service object.* We speak of a *service* as a service class or type (e.g., the file service), and a *service object* as an instance of the type (e.g., a file). Recent microkernel operating systems (e.g., Mach, Chorus, and Amoeba) represent all system abstractions as service objects named by capabilities.

Systems that support protected service objects are *extensible* because they make it easy for applications to define new services, simply by creating objects and passing capabilities to clients. Useful properties are automatically inherited by each new

Figure 4: An Opal Protected Pointer

service without the implementor reinventing mechanisms for these properties (e.g., object-grain access control and uniform naming across a network). In addition, capability-based protection limits the trust that a client must place in the implementor of a service. When the client invokes a service, it passes capabilities for other objects (e.g., segments) that it wants the service to operate on, and only those objects. The client trusts the service to behave correctly, but if the service fails it is restricted in the damage that it can do to the client. That is, protected services have no inherent privilege; in particular they do not run as "superuser". This use of object-orientation was originally emphasized in Hydra [Wulf 75] and capability-based computer architectures (surveyed in [Levy 84]).

5.1 Protected Pointers

An Opal service object is simply a language object that is internal to a *server* protection domain. By "internal" we mean that the object resides in a segment that the server has permission to attach, but the client domains do not; they can only operate on the object by making RPC calls to the server. Clients name service objects with *protected pointers*, based on capabilities in the Amoeba system (Chorus uses a similar scheme). An Opal protected pointer is a 256-bit quantity composed of four 64-bit fields, only three of which are important for the purposes of this paper (Figure 4):

- The name of a *portal* in the server domain. Portals are described in Section 2.1.
- The virtual address of the service object.
- A 64-bit *check field* whose value is chosen by the server, and allows the server to validate the protected pointer by matching with a corresponding check field stored with the object's representation.

A thread in a client invokes a service object by making an RPC call through the portal named in the protected pointer, passing the pointer and a procedure index (see Section 3) as arguments. This causes the thread to begin executing within the server domain at a virtual address chosen by the server, typically the first instruction of a standard code sequence to validate the check field and invoke the service object by making a procedure call through the appropriate entry in its dynamic table.

Protected pointers are difficult to forge because the value of the check field is probabilistically impossible for a client to guess, and there is no way to determine if a guess is correct except by asking the server. For this reason they are sometimes called *sparse capabilities* or *password capabilities*, but we differentiate them from true capabilities for the following reasons:

- They are not *impossible* to forge.
- They can be revoked by changing the value of the check field.
- The server may choose not to grant access even if a client presents a valid protected pointer.

For the purposes of this paper, the key difference from Amoeba capabilities is that the service object name embedded in the capability is a virtual address rather than an object surrogate that must be translated by the server. In addition, the naming and semantics of Opal portals differs from the corresponding notion of a *port* in Amoeba. The uniqueness of Opal portal names is assured by allocating them from the global address space, whereas Amoeba chooses its port names randomly from a large space, making collisions unlikely but not impossible. Furthermore, the node serving a given

Opal portal can be determined by calling the address space service, which is assumed to be secure and reliable. A node cannot masquerade as the server for a portal; if a compromised node claims to serve portals from a range of the address space that has not been assigned to it, other nodes simply will not believe it. In contrast, Amoeba locates server nodes by broadcasting their names on the network; nodes are prevented from illegally responding to such a request through the use of cryptographic techniques that require a physically secure network and special hardware support in the network interface.

5.2 On the Duality of Data and Services

Some systems treat service objects as special and distinct from language objects. For example, in early object-oriented operating systems, notably Clouds [Allchin & McKendry 83] and Eden [Almes et al. 85], service objects are *coarse-grained*, meaning that they are one-to-one with protection domains. More recently, Mach has supported protected objects directly in the kernel through its *port* abstraction. Like an Amoeba port, a Mach port is a message queue that clients send messages to and a server receives messages from. However, Mach views service objects as one-to-one with ports; a Mach capability is a handle to a kernel data structure that enables the holder to send messages to a particular port.[2] A Mach server allocates a port for each protected object it serves, and interprets all messages posted to that port as calls to the object; a client cannot call the object unless it holds a capability to send a message to the object's port.

Protection domains and communication endpoints (ports or portals) are heavyweight entities because they must be created and destroyed by the kernel. We envision systems with large numbers of service objects, and therefore reject approaches that use these abstractions directly to represent service objects. Furthermore, these kernel-based schemes force protection domains to call the kernel in order to exchange capabilities; in particular, capabilities cannot be stored in shared memory.

[2]In the Mach terminology, a capability is referred to as a "send port". Note that a send port is not itself a port, but only a name for a port (i.e., any number of send ports can exist for a given message queue).

In contrast to these systems, Opal makes no distinction between data and services; an Opal object is a "protected object" only from the perspective of a client that is insufficiently privileged to attach the object and invoke it with an ordinary procedure call. Any number of service objects may be clustered within the same protection domain and served through the same portal, reducing the cost of protection and communication. Like ordinary pointers, protected pointers are context-independent and can be passed between domains in shared memory (e.g., as arguments to a lightweight RPC call). Any domain that knows the value of a protected pointer can use it.

Note that kernel-controlled capabilities have some advantages over the approach used in Opal, Amoeba, and Chorus: kernel-controlled capabilities cannot be forged or stolen, and they can be reference-counted or garbage-collected by the operating system.

5.3 Persistent Services

Opal supports *persistent* service objects whose capabilities can be stored on disk and used after a system restart. This only requires that the system allow servers to restart and reregister the same portal names. As previously stated, portal names are allocated from the global address space and never reused, making them unique in space and time. Permission to reregister a portal name is itself represented by a capability; a privileged service (the *Portal Service*) maintains records of allocated portal names. The Portal Service restarts an idle server when a client attempts to call through one of its portals; to do this, it creates a fresh protection domain and restores the portals and segments that were active in the server at the time of the crash or shutdown. It is the server's responsibility to ensure that its segments are properly recoverable.

Given this facility, a service can be made persistent by making its service object(s) persistent. Opal's uniform address space ensures stable virtual address bindings for persistent data objects; therefore bindings for protected pointers to service objects are stable as well. Since persistent objects are activated by simple page faults, stored service objects are fetched from disk as a side effect of validating the object's check field, with no special action on the part of the server. In contrast, Mach capabilities are invalidated by a system

restart; Mach servers with persistent data must implement a secondary naming scheme for persistent objects, and do not benefit from the access control and reference-counting provided by Mach capabilities.

5.4 Proxies

Creation and use of shared service objects should be integrated with programming languages, just as shared data. However, most systems do not *directly* support transparent access to services, nor should they: in capability-based computer architectures, *all* objects are named by protected capabilities, making shared data as expensive as shared services. In Opal, as in most systems today, capabilities look different from data object pointers, and the client must handle RPC calls differently from ordinary procedure calls. Thus data objects are cheap and service objects are protected.

The details of accessing a service can be hidden from the application with language and runtime support for *proxy* objects [Shapiro 86]. A proxy is a local data object in the client whose interface is identical to the service interface, but whose methods are marshaling stubs.[3] A similar proxy exists on the server side to unpack and validate arguments to a service object call; the object address in a protected pointer is actually the address of the server-side proxy, rather than of the protected object itself. This organization is depicted in Figure 5.

Note that a protected object may have several proxies on the server side, each with a different check field. For example, multiple proxies can be used to represent different sets of access rights for the same object.

5.5 Summary

This section described how the mechanisms for shared and persistent data outlined in Section 4 are extended to encompass shared and persistent services named by *protected pointers* (capabilities). Opal has a unified view of data and services; service objects are simply language objects invoked through a *portal*. Any domain can act as a server for any objects attached to it. A server can export a new service simply by creating an object and passing a protected pointer to a client.

Services in Opal can be persistent, and protected pointers can be passed between domains in shared and persistent memory. Persistence of services derives directly from persistence of virtual storage. Similarly, the context-independence of protected pointers derives directly from the same property in the global address space.

6 Conclusions

We have described Opal, an operating system with a persistent, distributed virtual address space. We have outlined an object sharing package above this uniform address space that supports three independent and composable properties for language objects:

- *Sharing.* A single copy of an object can be shared between applications running in separate protection domains, either concurrently, or sequentially by transferring segment permission from one domain to another (object migration).
- *Persistence.* Objects can be stored on disk and retrieved transparently on pointer dereference.
- *Protection.* A server can construct *protected pointers* for its internal objects and pass them to its clients. A protected pointer allows the client to operate on the object by making protected RPC calls to the server, but not to attach the object directly.

The key point we wish to make is that there is no explicit kernel support for these object properties, *nor should there be.* Objects are a *policy* in Opal; their semantics and structure are imposed by a programming language and are unknown to the operating system kernel. This allows multiple object models to coexist and new object models to be developed. Nobody pays for features they do not want.

Furthermore, we assert that this approach promises the best performance. Opal objects are

[3] Opal's uniform address space also accommodates Shapiro's more general notion of a proxy, in which the server chooses the code that runs in the proxy. The server simply passes a pointer to the proxy type along with the capability. Of course, the client must trust the server to execute arbitrary code in the client's protection domain.

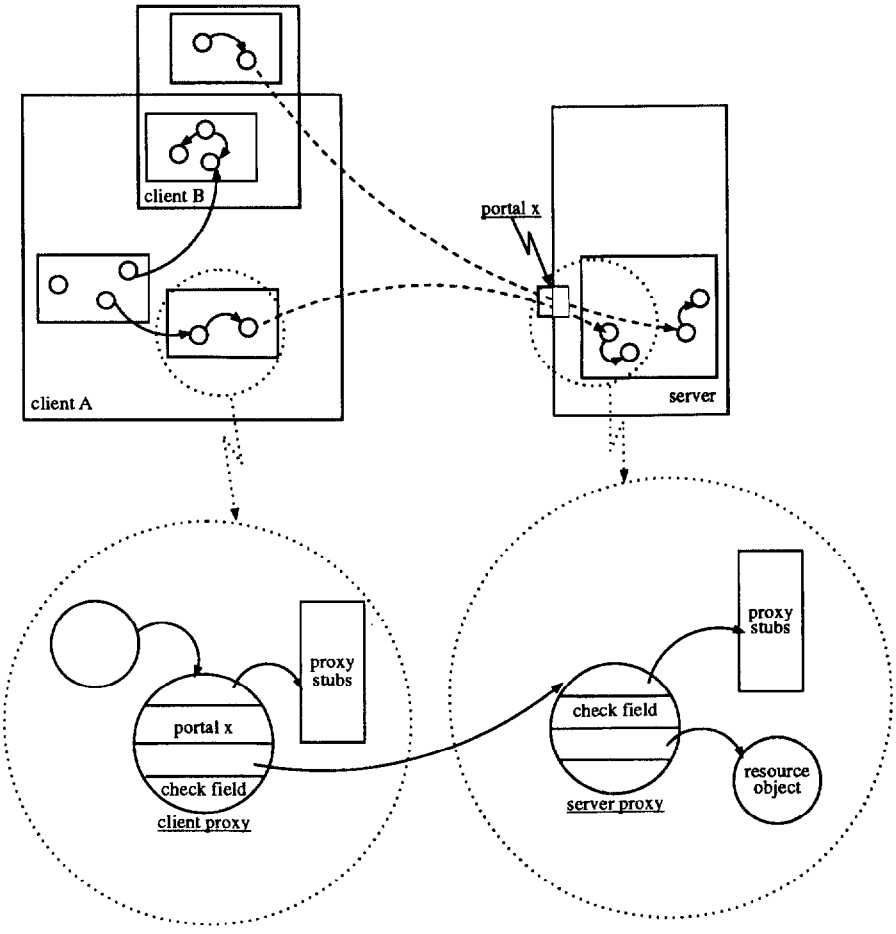

Figure 5: Service Objects and Proxies.

created and deleted without kernel involvement; object references can be passed and shared without kernel involvement. The fundamental mechanisms for accommodating object sharing are shared memory, page-based memory protection, virtual page faulting, lightweight user-level concurrency and synchronization, and high-performance RPC. The performance behavior of all of these features has been intensively studied and optimized.

To summarize, here are the elements of our philosophy for building object sharing systems:

- *Bring the language down to the system rather than bringing the system up to the language.* The kernel interface is not the programmer interface. Implement object semantics in the language and runtime system unless the operating system forces you to do otherwise.

- *Cluster objects to improve performance whenever possible.* Our implementation uses clustering in the following ways: (1) shared and persistent data objects are grouped into *segments* that are the granularity of access control, (2) passive persistent objects are faulted into memory in groups of virtual pages, (3) service objects that are related can be served from the same protection domain, and (4) service objects in the same server can be multiplexed through a single RPC communication channel.

- *Use virtual memory.* Fetch data with page faults. Let the virtual memory system manage object caching. Use shared memory whenever possible.

- *Do not constrain application use of protection boundaries.* Determine use of hardware-enforced protection according to which domain is invoking an object, not the object itself. Use cheap protection from a language type system whenever possible.

The recurring theme in all of these points is that support for object sharing and persistence should be built as an application-level runtime package that imposes object structure on untyped virtual storage. This is difficult to achieve because the virtual storage model presented by most operating systems is nonuniform; the virtual addresses used to name storage vary according to the physical location of that storage and which domain is referencing it, and there are many possible interpretations of a given virtual address. We believe that a uniform virtual storage abstraction, made possible by emerging 64-bit architectures, will have far-reaching implications for object sharing systems.

7 Acknowledgements

Ashutosh Tiwary reviewed early drafts and contributed greatly to the organization of the paper. Participants in the prototype effort include Mike Feeley, Valérie Issarny, and Ted Romer, who have also provided valuable comments. Chris Hebert helped with the diagrams. Thanks also to Brian Bershad, Craig Chambers, Eliot Moss, and Marc Shapiro for carefully delivered criticisms.

References

[Abrossimov et al. 90] V. Abrossimov, S. Habert, and L. Mosseri. COOL: Kernel support for object-oriented environments. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1990.

[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. 91] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support of the user-level management of parallelism. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 95–109, October 1991.

[Appel et al. 88] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent garbage collection on stock multiprocessors. *SIGPLAN Notices (Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation)*, 23(7), July 1988.

[Bal & Tanenbaum 88] H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. In *Proceedings of the International Conference on Computer Languages*, pages 82–91, October 1988.

[Bershad 90] B. N. Bershad. *High Performance Cross-Address Space Communication.* PhD dissertation, University of Washington, June 1990. Department of Computer Science and Engineering Technical Report 90-06-02.

[Cahill 88] V. Cahill. OISIN, the design of a distributed object-oriented kernel for COMANDOS. Master's thesis, Department of Computer Science, Trinity College Dublin, 1988.

[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. 89] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Little-

field. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 147–158, December 1989.

[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, February 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.

[Copeland et al. 90] G. Copeland, M. Franklin, and G. Weikum. Uniform object management. In F. Bancilon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology - International Conference on Extending Database Technology 1990 (Lecture Notes in Computer Science 416)*, pages 253–268. Springer-Verlag, 1990.

[Daley & Dennis 68] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[Detlefs et al. 88] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, December 1988.

[Dig 92] Digital Equipment Corporation, Maynard, MA. *Alpha Architecture Handbook*, 1992.

[Eppinger 89] J. L. Eppinger. *Virtual Memory Management for Transaction Processing Systems.* PhD dissertation, Carnegie Mellon University, February 1989. CMU-CS-89-115.

[Fabry 74] R. S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[Faust & Levy 90] J. E. Faust and H. M. Levy. The performance of an object-oriented threads package. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1990.

[Feeley & Levy 92] M. J. Feeley and H. M. Levy. Distributed shared memory with versioned objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1992. University of Washington CSE Technical Report 92-03-01.

[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.

[Khoshafian & Copeland 86] S. N. Khoshafian and G. Copeland. Object identity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 406–416, 1986.

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

[Li 86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD dissertation, Yale University, September 1986. YALEU/DCS/RR-492.

[Liskov et al. 87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 111–122, November 1987.

[Marques & Guedes 89] J. A. Marques and P. Guedes. Extending the operating system to support an object-oriented environment. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 113–122, September 1989.

[MIP 91] MIPS Computer Systems, Inc., Sunnyvale, CA. *MIPS R4000 Microprocessor User's Manual*, first edition, 1991.

[Moss 90] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. COINS Object-Oriented Systems Laboratory Technical Report 90-38, University of Massachusetts at Amherst, May 1990.

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

[Rozier et al. 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, P. Leonard, S. Langlois, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4), 1988.

[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.

[Shapiro et al. 89] M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for C++ objects. In *Proceedings of the Third European Conference on Object-Oriented Programming*, July 1989.

[Wilson 91] P. R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *ACM SIGARCH Computer Architecture News*, 19(4), June 1991. University of Illinois at Chicago Technical Report UIC-EECS-90-6, December 1990.

[Wulf 75] W. A. Wulf. Overview of the Hydra operating system. In *Proceedings of the 5th Symposium on Operating Systems Principles*, pages 122–131, November 1975.

[Young et al. 87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.