The Alpha Operating System: Programming Model

J. Duane Northcutt
Raymond K. Clark

TR #88021

Archons Project Technical Report

February 15, 1988

The Alpha Operating System: Programming Model

J. Duane Northcutt
Raymond K. Clark

February 15, 1988

Archons Project Technical Report #88021

This research was supported in part by the USAF Rome Air Development Center under contract number F30602-85-C-0274. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of RADC or the U.S. Government.

.

Table of Contents

List of Figures

List of Tables

Abstract

Alpha is an adaptable decentralized operating system for real-time applications, being developed as a part of the Archons project's on-going research into real-time distributed systems. Alpha is unique in two particularly significant ways—first, it is *decentralized,* providing reliable resource management transparently across physically dispersed nodes, so that a distributed application can be implemented and executed as though it were centralized; and second, it provides comprehensive, high technology support for real-time applications, particularly supervisory control systems (e.g., industrial automation applications) which are characterized by predominately aperiodic activities, executing under critical time constraints (such as deadlines). Alpha is extremely adaptable so as to be easily optimized for a wide range of problem-specific functionality, performance, and cost. Alpha is the first systems effort of the Archons Project, and the prototype is being created at Carnegie-Mellon University directly on Sun workstation hardware [Northcutt 88b]. It has been demonstrated with a real-time control application written by its first industrial user, General Dynamics [Clark 88a]. While the second-generation, commercial-quality version produced by Kendall Square Research is portable, its initial target is their new multiprocessor. Both versions of Alpha are sponsored by the USAF Rome Air Development Center and are in the public domain for U.S. Government use.

The focus of this report is on the conceptual programming model presented (primarily) by the kernel layer of the Alpha operating system. The intent is to describe the native programming paradigm of the kernel, without delving into the specific details of the kernel's programming interface or into issues related to the various client-specifiable system functions (e.g., virtual memory management policies, device management, and I/O handling).

This report describes the functionality of the kernel of the Alpha operating system as seen by a programmer. It begins by briefly defining the context and rationale for this work, the overall technical approach taken, and the key features of the system. The basic programming abstractions provided by Alpha are then described, followed by a description of the ancillary abstractions that serve to complete the programming model. Finally, the Alpha programming model is compared and contrasted with more conventional programming models that are in use today.

1 Introduction

Alpha is an adaptable decentralized operating system designed to provide reliable resource management transparently across physically dispersed nodes, in support of real-time applications. The real-time applications of particular interest for Alpha are supervisory control systems (e.g., industrial automation applications), which are characterized by predominately aperiodic activities, executing under critical time constraints (such as deadlines). Alpha has furthermore been designed to be extremely adaptable so as to be easily optimized over the wide range of problem-specific functionality, performance, and cost requirements commonly associated with real-time applications.

The focus of this report is on the conceptual programming model presented (primarily) by the kernel layer of the Alpha operating system. The intent is to describe the native programming paradigm of the kernel, without delving into the specific details of the kernel's programming interface (which is described in [Northcutt 88d] and [Shipman 88]), or into issues related to the various client-specifiable system functions (e.g., virtual memory management policies, device management, and I/O handling). This report was written from the perspective of the builder of Alpha's kernel layer, and throughout the remainder of this document all references to Alpha should be interpreted as meaning the kernel of the Alpha operating system.

The following sections describe the research context in which Alpha was developed, the general technical approach taken to the system design and implementation effort, and the key features of the system.

1.1 Research Context

The Alpha programming model was derived from the requirements defined for the Alpha operating system's chosen application domain—i.e., distributed real-time command and control applications. A unique combination of requirements is implied by this application domain, and existing systems either ignore some of the requirements or fail to provide a comprehensive, well-integrated solution. The Alpha programming model provides features that meet each of the main requirements of the defined application domain.

Some of the requirements adopted for Alpha are unique to the supervisory control context, while others, although generally applicable to a wide range of systems, are especially important in this domain. These requirements can be grouped into four categories:

- timeliness requirements,
- physical distribution requirements,
- robustness requirements, and
- adaptability requirements

In addition to the examination of the problem area's requirements, the programming model that was developed for Alpha stems from an in-depth study of several key techniques, often involving the synthesis of new concepts for solving fundamental problems. The approach taken explored both hardware and software alternatives in order to effectively implement the basic system concepts.

This research resulted in the creation of a set of programming abstractions that are intrinsically well-suited to modular, reliable, decentralized operating systems, along with the design and implementation of a set of kernel-level mechanisms to support them. Alpha is not a monocentric system, featuring one particular concept over all others, nor is it a collection of facilities grafted onto an existing operating system. All of the system's features were designed based on the requirements of the application domain and implemented in concert with each other, resulting in a well-integrated solution within a meaningful framework.

Alpha differs from other operating system efforts in a number of ways. Alpha was designed in a top-down fashion based on a high-level requirements, not bottom-up based on a set of implementation details†. Furthermore, Alpha was not constrained to be compatible with existing (system or application) software, and so the exploration of the major research concepts was not compromised by having to force newly developed features into an existing system context. For this reason, the Alpha operating system integrates concepts in the most effective way to achieve its objectives. If the Alpha operating system's requirements included a constraint such as UNIX compatibility, many of the system's trade-offs would be made differently and compromises would undoubtedly have to be made in the conceptual fabric of the system. Alpha represents the results of unconstrained development of a series of (application-requirements-driven) concepts, and not the force-fitting of ideas into some incompatible, artifact-ridden structure.

Alpha was designed to be executed on architectures consisting of a set of loosely-coupled nodes, where each node could be either a uniprocessor or a multiprocessor. Additional constraints placed on the implementation of Alpha dictate that the system must be able to run on standard, off-the-shelf processors (i.e., the system cannot be dependent on specialized hardware support to make its implementation practical) and that the system's native programming model must be similar in concept and implementation to traditional models (i.e., the system cannot demand that its programmers learn and adopt a radically different approach to programming).

Complete documentation of the context, assumptions, objectives, and rationale for this effort is to be found in [Northcutt 88a].

1.2 Technical Approach

This report describes the facilities provided by Alpha to a programmer at its native kernel interface. The Alpha kernel provides a set of simple and uniform programming abstractions from which modular, reliable, and distributed real-time control applications may be constructed. The purpose of a kernel is to provide fundamental abstractions and mechanisms that support a range of different system interfaces (i.e., operating systems and languages, similar to [Habermann 76]), which is not the same as a trivial operating system or an executive. The Alpha kernel mechanisms have been carefully and deliberately constructed to be devoid of all policy decisions, and are meant to support the exploration of a wide range of decentralized operating system policies.

The interface provided by the kernel is not necessarily the same interface that the Alpha operating system presents to its application programmers. In most cases, there is a system layer on top of the kernel that can provide the applications programmer with a high-level programming language, or a more abstract system interface (possibly independent of any specific programming language). The client programmer of the Alpha kernel could be: a systems programmer involved in the construction of the higher levels of the Alpha operating system; the creator of a run-time support package for some standard, high-level programming language to be used

†*Of course, no design proceeds in a purely top-down manner, oblivious to the realistic underlying capabilities of a system. The point here is simply that we have attempted to build what we want, not what we already know how to build.*

by applications programmers; or an applications programmer creating an embedded application by programming directly on the kernel's interface.

The abstractions provided by the kernel of Alpha are based on a combination of the principles of object-orientation [Bayer 79], atomic transactions [Bayer 79], replication [Randell 78], and decentralized real-time control [Jensen 76a].

The general programming paradigm supported by the kernel of the Alpha operating system is known as *object-oriented programming* [Goldberg 83, Cox 86], and the primary abstractions supported by the kernel are: *objects*, *invocations*, and *threads*. In Alpha, *objects* adhere to the common definition of abstract data types and interact with other objects via the *invocation* of *operations* on them. *Threads* are the manifestations of control activity (i.e., the units of concurrent computation and scheduling) within the kernel.

The Alpha thread abstraction is in many ways comparable to the process abstraction found in many conventional systems. Unlike conventional processes however, threads move among objects via invocations, and do so without regard for the physical node boundaries of the system. Furthermore, the kernel's object abstraction extends to all system services and devices—i.e., Alpha encapsulates all of the system's physical resources within standard object interfaces.

1.3 System Structure

The following is a brief description of the structure of the software and hardware in the initial implementation of Alpha. The intent of these descriptions is to provide an overview of the framework in which the Alpha kernel exists. A detailed description of the structure of the Alpha system software can be found in [Northcutt 88c]; details of the system's hardware structure can be found in [Northcutt 88b].

1.3.1 Software

The Alpha operating system lies between the application code and the hardware and (in its current implementation) has the internal, logical structure shown in Figure 1. The *application* layer consists of a collection of objects and threads, defined by one or more applications programmers. These objects and threads execute without any special system privilege (with respect to priorities, access to internals, etc.). The separation of application-level object address spaces is enforced by the underlying hardware.

The *system* layer consists of a collection of objects and threads, identical to those found in the application layer (i.e., unprivileged with hardware-enforced separation of address spaces). System-level objects and threads are distinguished from their application-level counterparts only by their access privileges—i.e., the *capabilities* that they possess (see Section 3.1 for an explanation of the access control mechanisms). Objects in the system layer provide the higher-level operating system services—e.g., name, authentication, directory, and reconfiguration servers, as well as user interface and programming environment support facilities.

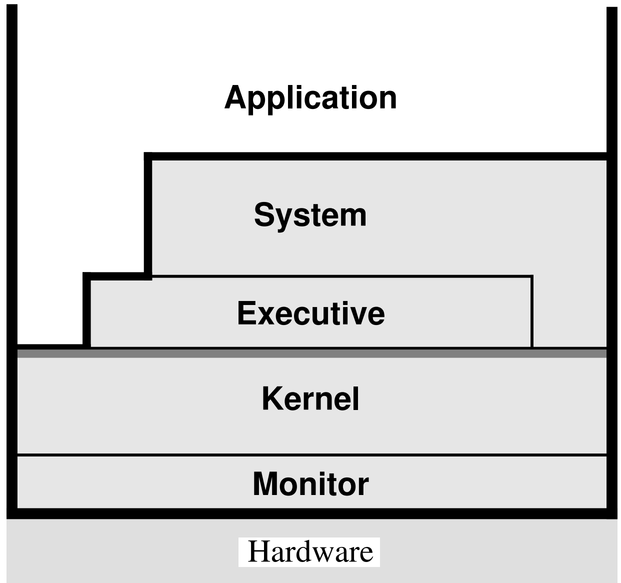

Figure 1: Logical System Software Structure

The *executive* layer extends the functionality of the kernel and is composed of a collection of special entities known as *kernel objects* and *kernel threads*. These appear to the programmer who specifies them as normal objects and threads, but they have special characteristics and are implemented in a different manner. Kernel threads and kernel objects exploit optimizations that permit the system-builder to easily migrate a limited number of threads and objects into the lower layers of the system, primarily for reasons of performance (as well as access to kernel or device shared data). Kernel object types are specified in the same way as normal objects, however kernel object instances are linked into the system at system-build time and coexist within the kernel's address space. Kernel threads and objects are used to implement system daemons (e.g., interrupt handlers, virtual memory pagers, transaction managers, and garbage collectors). In addition, the executive layer is where the policy modules that govern the behavior of the local kernel mechanisms reside.

The *kernel* layer provides the basic system abstractions upon which the remaining portions of the system are constructed. The kernel consists of:

- system service objects—i.e., routines that present object interfaces to the client and provide essential services (e.g., object and thread managers, and semaphore and lock managers)
- kernel subsystems—i.e., collections of routines that collectively provide a major internal system facility (e.g., virtual memory, scheduling, communications, and secondary storage)†.
- the kernel proper—i.e., routines the provide the bulk of the kernel's basic functionality (e.g., objects, threads, operation invocation, trap handling, subsystem interfaces, and physical memory and I/O peripheral interfaces)

†*Many of these facilities are executed, wholly or in part, on processors separate from the one on which the remainder of the kernel executes.*

Finally, the *monitor* layer consists of software that exists within each processor's on-board PROM storage. Included in the functions provided by the monitor layer are: low-level I/O support (including the "printf()" routine); TFTP-based [Sollins 81] boot, upload, and download support; power-on reset initialization and diagnostics; and low-level debugging support.

1.3.2 Hardware

The hardware base on which the Alpha operating system executes consists of a loosely-coupled collection of dedicated-function multiprocessor nodes, constructed from largely off-the-shelf system components. The testbed supports the development of software by a collection of system programmers working from individual (remote) workstations. Furthermore, the nodes of the testbed allow the exploration of various operating system concepts that may benefit from hardware support.

At a high level of abstraction, the testbed consists of three components—the *development and control system*, the *distributed computer system*, and the *application system* (see Figure 2). The development and control system consists of a collection of Sun Microsystems workstations, running the UNIX operating system, that are connected to the distributed computer system via both Ethernet and 9600 baud serial lines. This part of the testbed is used to develop application programs, to control applications running on the distributed computer system, and to monitor the operating system as well as the application during experiments. The software tools that exist on the Sun Microsystems workstations (editors, compilers, linkers, "make", window managers, etc.) provide an effective development environment, permitting the testing and debugging of low-level system code by multiple, remote users.

The testbed's distributed computer system is the host on which the Alpha operating system executes. It consists of a collection of processing nodes interconnected by a global communications subnetwork. The distributed computer system is logically a single computer, where the network is analogous to the backplane and the nodes correspond to the cards of a conventional computer. The distributed computer is partionable into separate computers, and interfaces with the outside world via a gateway machine (i.e., a standard Sun Workstation running UNIX).

The application system is comprised of a set of application devices, and is the interface between the distributed computer and the physical system it is controlling. Application devices can be sources of data to the computer (e.g., sensors) or data sinks (e.g., actuators), and they can be a combination of both actual and simulated devices. The application system interfaces to the distributed computer system in a variety of ways—directly through nodes, through the network, or through gateway machines. All of these cases are illustrated in Figure 2.

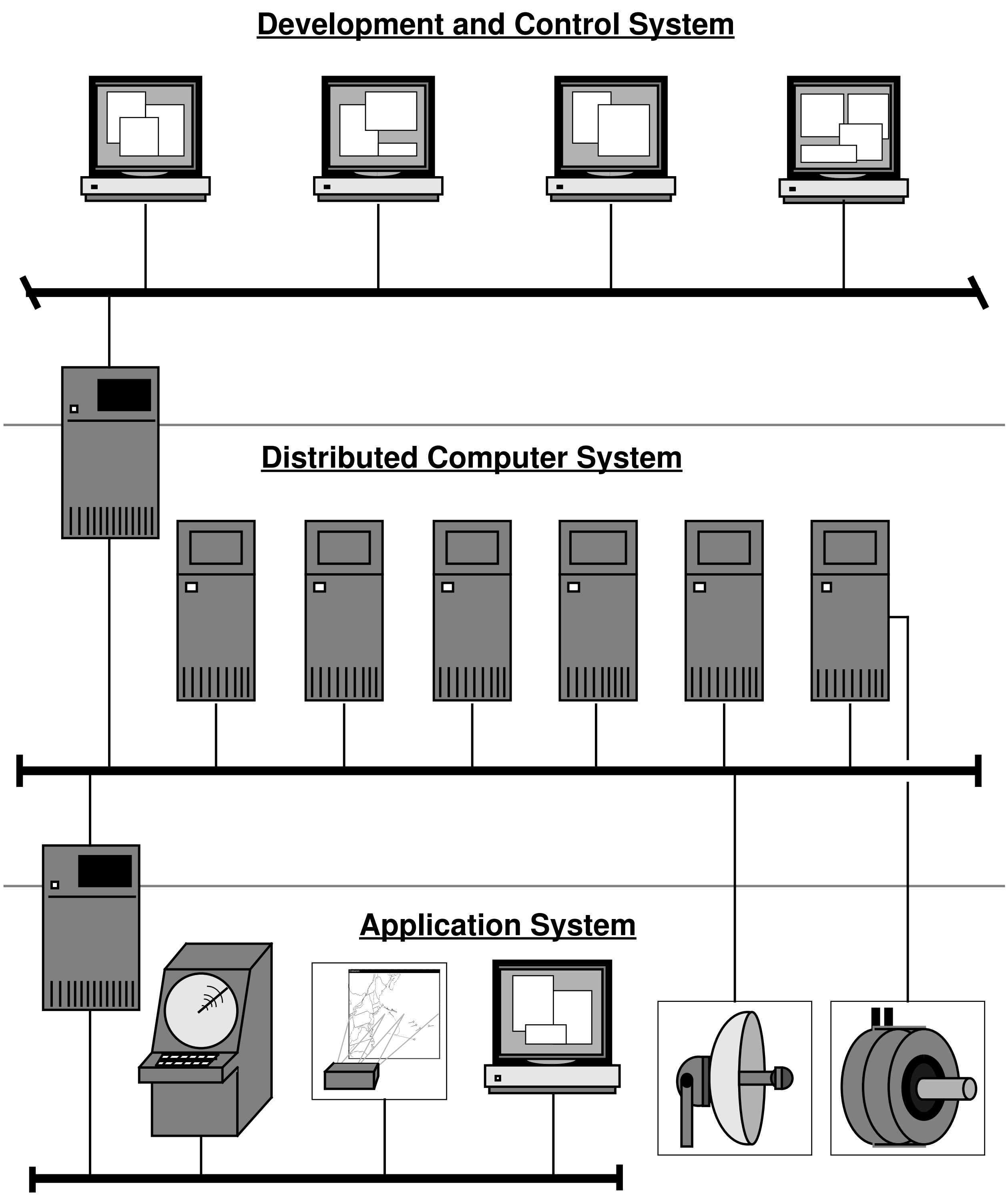

Figure 2: Testbed System Structure

1.4 Key System Features

Alpha was designed, from the hardware up, to support application domain requirements, as defined in [Northcutt 88a], concerning timeliness, distribution, robustness, and adaptability. Concern for these requirements permeated the design and implementation process of Alpha, and the effects of this are perceptible throughout the system. The following subsections briefly describe these requirements areas and the effects they had on the Alpha programming model.

1.4.1 Timeliness

The most visible manifestations of the concern for timeliness in the design and implementation of Alpha is the use of time-driven resource management based on application-specified time constraints and resource management policies. Alpha uses a novel and highly effective technique to explicitly express an application's time constraints—as the time-dependent value to the system of completing specified computations—in a straightforward and natural fashion. A guiding principle in the design of Alpha was that all resource management decisions should be made based on the time constraints of the entity making the request for system resources, and when all of these requests cannot be met in a timely fashion, an application-specific overload handling policy should be followed to deal with the requests. To this end, the thread abstraction was developed as the fundamental activity with which to associate time constraints (which, in turn, are used to accomplish global, time-driven resource management).

The time-driven management of system resources in Alpha depends on the correspondence between the programmer's and system's view of application computations (provided by threads), and application-specified importance and time constraint information (provided by attributes of threads). Threads represent the programmer's logical view of each concurrent stream of execution that makes up an application, and the system's physical manifestation of these logical computations. Thus, threads provide a direct means for associating the timeliness requirements that clients specify for their computations, with the specific, run-time entities that the kernel manages. In this manner, global importance and urgency characteristics of computations can be propagated throughout the system and used in resolving contention for system resources according to client-defined policies.

The thread abstraction in Alpha was not designed with only timeliness in mind—distri-bution, robustness, and adaptability requirements were also considered. Threads provide a unified means of managing all resources in the system, both within and among the processing nodes in the distributed system. The application-specified timeliness attributes of computations are carried with threads as they move through objects and across the system's nodes. This allows a globally consistent form of distributed resource management to be provided through a form of implicit coordination: at each node, the same resource management policies are applied to the threads' global attribute information.

1.4.2 Distribution

Alpha was designed to execute on a physically distributed computing system because real-time process control applications tend to be physically distributed by their very nature, as well as for reasons of robustness and performance. To obtain the benefits offered by distributed systems, a number of functions must be performed and problems must be addressed that either did not exist in centralized systems or could safely be ignored. In general, either the system software or the application programmer must perform inter-node resource management and deal with the effects of physical distribution.

The Alpha operating system deals explicitly with the effects of distribution by providing, for example, a facility for efficiently providing reliable, physical-location-transparent communication at a low level in the system—i.e., the operation invocation facility. The operation invocation facility is the primary kernel service upon which all other abstractions depend. By making the invocation of operations on objects reliable and location-transparent, the effects of physical distribution are reduced, in effect, to the semantics of procedure calls that return an indication of the success or failure of the invoked operation. Furthermore, by having all objects (and even mechanisms within the kernel) use the invocation mechanism, it is possible to enforce a uniform access method to all system resources, regardless of their actual location within the system. System resources can therefore be managed and accessed uniformly regardless of their physical location.

The operation invocation facility represents the single focal point of all interactions among objects, as well as between objects and the kernel. This provides a convenient point where system access control and data format translation functions can be performed. Alpha has attempted to eliminate any alternate communications channels (such as shared memory) that might be used by programmers and would inhibit the system's ability to perform dynamic reconfiguration. Also, the full visibility of all instances of inter-object communication makes possible a number of system resource management optimizations that make use of object/thread interaction patterns.

In addition to the operation invocation facility, many of the Alpha operating system's other mechanisms are designed to cope with the effects of the system's physical distribution, including mechanisms to dynamically (and transparently) migrate objects among nodes.

1.4.3 Robustness

The robustness techniques employed in Alpha are supported primarily by kernel mechanisms that provide a client interface at which failures in the underlying system are abstracted into a set of well-defined, predictable behaviors. In particular, the following robustness goals are addressed: consistent behavior of actions, availability of services and data, graceful degradation, and fault containment.

While the Alpha operating system provides a set of mechanisms to support these objectives, its robustness mechanisms are not intended to form a wholly self-contained facility. Rather, the kernel is intended as a framework within which policy issues relating to these robustness techniques can be explored.

The Alpha operating system's concern for reliability is manifest at all levels within the system—from the basic assumptions, to the programming abstractions, and all the way down through the system's design and implementation. The variety of object-orientation in Alpha was chosen in the belief that it would be well-suited to the type of robustness techniques that have been developed by the Archons project for real-time command and control applications [Clark 88b, Sha 85]. The Alpha object model provides a stylized control structure for interactions among software components that is more manageable than those of the common process and message-based system model, and that prevents unrestricted access to data. Among other things, this serves to simplify the task of tracking the operations performed on distributed data on behalf of atomic transactions. The fact that the object model centralizes all access to encapsulated data reduces the complexity involved in structuring operations so as to maximize the concurrency that can be obtained from objects (both within and outside of atomic transactions). Furthermore, it is possible to make incremental changes to objects with increased hopes of correctness because the code that manipulates specific data items is centralized within an encapsulating object.

The Alpha kernel provides mechanisms for handling a wide range of user- and system-defined exceptions in a manner consistent with the system's objectives and programming model. For example, because of the distributed nature of Alpha, processing nodes may fail while threads span nodes. This means that it is both possible for an invocation to fail, and for portions of the computations to become detached (or *orphaned*). Therefore, Alpha provides a means of indicating the failure of operation invocations, as well as a means for detecting and eliminating orphan threads in a timely fashion. Furthermore, Alpha provides a means of dealing with the exceptions that stem from the inability of a computation to meet its given execution time constraints. Handling these exceptions requires the same prompt attention as is needed for the elimination of orphaned threads. In addition, atomic transactions require similar thread manipulation mechanisms to handle exceptions related to the abortion of transactions—due to either a node failure or an explicit abort command.

To manage these exceptions (as well as machine- and user-defined exceptions), Alpha provides a mechanism to divert a thread's control to an appropriate exception handler whenever an exception occurs. Furthermore, this exception handling mechanism preserves the state of the thread that incurs the exception in order to permit the graceful recovery of a computation following the exception. When a thread encounters an exception in Alpha, the objects affected are returned to a consistent state by executing the exception handlers for each block of code that the thread is active within when the exception occurs. This allows a thread to perform its own clean-up operations, executing with its own proper attributes (e.g., time-constraints), while possibly spanning multiple objects and nodes.

The robustness of Alpha is enhanced by optimizing the design and implementation for the exception cases, instead of the expected cases. Examples of the application of this principle can be found in the system's exception handling mechanisms, communications protocols, and operation invocation facility. For instance, the kernel does not use hints in any form—e.g., the internal global identifiers used to access Alpha programming entities (e.g., threads and objects) do not include an explicit reference to the physical location of the referenced entity.

The Alpha operating system supports the graceful degradation of function through a collection of resource management facilities that make use of an ordering function (currently based on thread timeliness constraints and relative importance) associated with all requests for services, in order to sacrifice lower-valued requests in favor of higher-valued ones when resource allocation conflicts arise.

Alpha supports the objectives of fault containment by placing each object in a separate (hardware-enforced) address space, and by separating these software components into private system-enforced protection domains, with all interactions restricted to those explicitly allowed by the system's capability facility. This approach provides a degree of defensive protection, where errors are prevented from propagating among objects in an unconstrained fashion.

1.4.4 Adaptability

One of the reasons for choosing the form of distributed computer system upon which Alpha executes is the high degree of extensibility that is inherent in loosely coupled, bus-structured, distributed systems.

The adaptability of the system resource management facilities in Alpha is supported to a great extent through the use of (both static and dynamically acquired and applied) application-specified information. In particular, a wide range of application- and system-specific attributes can be associated with computations, embodied by threads, that are then carried along with the threads as they move through the system. The application-specified information allows the system's resource management algorithms to adapt to the changing demands placed on the system by the application as the availability of system resources changes.

In addition, the adaptability requirements of Alpha are addressed in two major ways—modularity for the operating system's clients is supported through the use of the object-oriented programming model supported by the operating system, and adaptability within the operating system itself is provided by the use of a policy/mechanism separation approach [Hansen 70].

The kernel provides a simple and uniform interface to its clients that centers around the operation invocation facility. The object programming abstraction supported by the operating system exhibits the same benefits associated with object-oriented programming abstractions in general, among which are information hiding, increased modularity, enhanced uniformity and simplicity of the programming interface, and reduced life-cycle costs [Bayer 79, Cox 86].

Alpha's kernel is implemented as a collection of mechanisms from which policy decisions were carefully excluded. Each major logical function in the kernel is manifest in an individual mechanism, and a great effort was made to ensure a proper separation of concerns among these mechanisms. Adaptability is achieved through the separation of functions into mechanisms; implementation changes are restricted to individual mechanisms, and changes in system policy do not require changes in the functionality of mechanisms, just changes in the use of mechanisms. In addition, most of the major subsystems consist primarily of a collection of mechanisms set into a framework upon which a wide range of differing policies can be imposed, allowing the easy addition or modification of resource management policies.

2 Basic Programming Abstractions

The Alpha kernel is based on a small set of basic mechanisms, similar to the those in Accent [Rashid 81]. The Accent kernel is based on the process and interprocess communication abstractions, while the Alpha kernel is based on the abstractions of objects, the invocation of operations on objects, and threads. As in Accent, where system calls are performed by sending messages to processes, all kernel services in Alpha are provided by the invocation of operations on objects.

Alpha implements an interface on top of the system hardware that provides the kernel's basic programming abstractions of objects, operation invocation, and threads. The following sections define each of these basic abstractions, and describe their characteristics and features.

2.1 Objects

At a high level of abstraction, the Alpha object model is similar to the common definition of simple abstract data types, as it is in other object-oriented operating systems (e.g., [Allchin 83], [Almes 85], [Liskov 84], or [Schantz 85]). In Alpha an object encapsulates data and provides a set of operations to manipulate that data. An object can only be accessed via the operations that constitute its interface (i.e., the object's operation entry points). Additionally, the operations specify the number and types of parameters that are to be passed into and out of the object when the operations are invoked.

The Alpha object model emphasizes a simple and uniform system interface, minimizing the specialized artifacts that are introduced into the (logical as well as physical) programming model—everything appears as an object to the programmer. In particular, the object abstraction in the Alpha kernel extends to all system services, and encapsulates all of the system's physical resources, thereby providing programmers with object interfaces to all system-managed resources (e.g., memory and devices). This uniform system interface allows operations to be invoked on a wide range of entities, ranging from user or system objects and threads, kernel routines, to system hardware.

2.1.1 Basic Definitions

Objects in the Alpha kernel are roughly equivalent to abstract data types—i.e., some encapsulated data along with the code for a set of operations with which the data is manipulated. Objects are written by programmers as independent modules, composed of the object's data and the operations that define its interface.

In Alpha, objects are passive entities; there is no activity within an object until an operation has been invoked on it. Upon operation invocation, an object becomes active—i.e., it executes the code associated with the invoked operation (which may, in turn, involve the invocation of operations on other objects). Once the operations invoked on an object complete, the object again becomes inactive, awaiting further invocations.

In a running Alpha system, all objects are *instances* of various object *types*. An object type is a template that defines the structure of an object, the initial values of the object's data, and the operations associated with the object. Object types are passive entities, defined by programmers and maintained within the system object store. New object type definitions can be dynamically created and added to the system at run-time†. Object instances are the executable, run-time manifestations of objects in Alpha that may be created and deleted dynamically. In the following discussion (except where explicitly noted otherwise), the term *object* refers to an instance of a specific object type.

In addition to the attributes of modularity, information-hiding, maintainability, etc. normally associated with an object-oriented programming paradigm [Cox 86], the programming model described here is especially well-suited to support decentralized, highly-concurrent implementations of the major robustness mechanisms in Alpha (i.e., atomic transactions and replicated objects). Because the basic abstractions of Alpha were developed to meet the system's special requirements, its definition of objects differs somewhat from more common object definitions. For this reason, insofar as possible, the names of the Alpha programming entities were chosen to avoid confusing references to existing object-oriented programming terminology (including the terms: *message*, *method*, and *class*).

Objects in Alpha are simple, passive entities, the main characteristics of which are the rigid encapsulation of state information and clear definition of an interface to the encapsulated information by a set of operations. To enforce the encapsulation of information within objects, each Alpha object exists in a separate address space [Lampson 69]. To enforce the integrity of the interface projected by an object, the kernel ensures that execution within an object can begin only at entry points corresponding to the object's operations, and these operations can only be accessed by employing the operation invocation facility. In addition, the only manner in which information can be exchanged among objects is via the explicit passing of parameters on operation invocation.

A simplified example of an object in Alpha is illustrated in Figure 3. In this example the object is a prototypical queue object named ***Queue***, with three client-defined operations: INITIALIZE, INSERT, and REMOVE. This queue object includes the data that make up the elements of the queue, the code that implements the operations, and other code and data that comprise the internal implementation of the object (e.g., storage for the queued elements, various utility subroutines, pointers used to keep track of the entries within the object, or data required for internal synchronization). This figure reveals the internal structure of the object—however, only the entry points defined at the interface are visible to the objects that use it.

Figure 4 provides an example of an object type specification for the sample queue object shown in Figure 3. This object is specified in the extended C programming language used by the Archons research project to write application programs for Alpha (see [Shipman 88] for details of this interim object programming language).

†*This reflects changes made to the original programming model as a part of the Alpha Release 2.0 work being done at Kendall Square Research.*

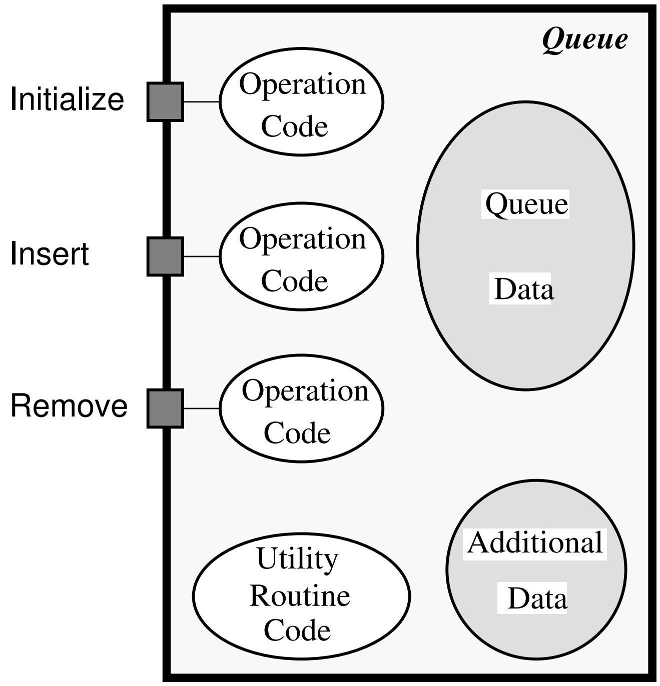

Figure 3: Example Object

```
/* An Example Queue Object Type Specification */
OBJECT QueueType() {

/* Declarations */
static boolean          qfull, qempty, init = FALSE;
static char             queue[Q_SIZE];
static int              qtail, qhead;

/* Operations */
OPERATION Initialize()
        /*
         * Initialize the queue object.  Initialize the queue pointers and flags,
         *  and then return successfully.
         */
        {
                /* initialize the queue pointers */
                qhead = qtail = 0;

                /* initialize the queue's status flags */
                qfull = FALSE;
                qempty = TRUE;

                /* indicate that the queue has been initialized */
                init = TRUE;
        };
```

Figure 4: Example Object Type Specification

```
OPERATION Insert(IN char: chr)
        /*
         * Take a character and insert it in the queue if it is not full.  If the queue is full, or has not
         *  been initialized, a failure is indicated, otherwise a 'SUCCESS' is returned.
         */
        {
                /* check if the queue has not been initialized or is full */
                if ((init != TRUE) || (qfull == TRUE)) RETURN FAILURE;

                /* insert a character into the queue */
                queue[qhead++] = chr;
                qhead %= Q_SIZE;

                /* check if the queue is now full */
                if (qhead == qtail) qfull = TRUE;

                /* indicate that the queue is not empty */
                qempty = FALSE;
        };


OPERATION Remove(OUT char: chr)
        /*
         * Remove a character from the queue, if it is not empty.  If the queue is empty, or has
         *  not been initialized, a failure is returned, otherwise a 'SUCCESS' indication is
         *  returned.
         */
        {
                /* check if the queue has been initialized */
                if (init != TRUE) RETURN FAILURE(QNOTINIT);

                /* check if the queue is empty */
                if (qempty == TRUE) RETURN FAILURE(QEMPTY);

                /* remove a character from the queue */
                chr = queue[qtail++];
                qtail %= Q_SIZE;

                /* check if the queue is now empty */
                if (qhead == qtail) qempty = TRUE;

                /* indicate that the queue is not full */
                qfull = FALSE;
        };

}; /* end of the queue object */
```

Figure 4, continued

2.1.2 Associated Features

Objects have a number of additional features associated with their use in Alpha. Some of these features are supported by kernel mechanisms, such as those that permit the manipulation of objects (e.g., their creation, deletion, and physical placement). Other features are extensions of the basic definition of objects, such as the standard operations which are defined on all objects by the system (in order to manipulate the internal representations of objects). Finally, some of these features are attributes associated with objects that affect the manner in which the system manages them.

It should also be noted that some of these features are not strictly a part of the logical programming model, but rather reflect the ways in which physical reality intrudes on a logical design.

2.1.2.1 Object Management

Objects are created and destroyed by invoking operations on a kernel-provided object management object. To create an object, an invocation is made on the create operation of the object manager. The parameters of this invocation include: a specification of the type of object to be created; the attributes that are to be (initially) associated with the newly created object; and the initialization parameters for the object being instantiated (e.g., maximum data size, object attributes, and replication factor). This operation invocation returns an identifier for the newly instantiated object, which is used in all subsequent references to the object. Objects are deleted from the system by invoking the delete operation on the object manager and passing, as an invocation parameter, the identifier for the object to be deleted. (See [Northcutt 88d] for a detailed description of the system-provided object management object).

An object is created on the node at which the create operation was invoked, and an object cannot be split across nodes (i.e., an object exists on a single node at any point in time). Furthermore, the kernel provides a mechanism with which a given object can be migrated to a specified node. With these kernel mechanisms, a wide range of policies for initial object placement and object migration can be implemented at the system level of Alpha.

While the Alpha system philosophy would have all operations performed on the target object itself, this is not possible for the object creation operation. It is for this reason that the kernel provides the object manager. Note that, while it would be possible to destroy an object by invoking a delete operation directly on it, for reasons of symmetry the delete operation is performed on the object manager instead.

2.1.2.2 Standard Operations

In addition to the client-specified operations that are defined by way of an object's type specification, all objects have a set of special, system-provided operations defined on them. These *standard operations* are automatically associated with all objects in order to allow the manipulation of the object's internal representation in a conceptually consistent, object-oriented manner.

The system-defined operations on objects provide the ability to:

- move an object among nodes
- suspend or continue the execution of threads within an object

- move the state of an object to and from its secondary storage image
- commit or abort the effects of the operations performed on an object by threads executing atomic transactions
- extend and contract an object's dynamically allocatable memory region

These operations are useful for such purposes as: debugging objects, dynamic object reconfiguration, applying atomic transaction semantics to objects, and maintaining a consistent representation of an object's state information across node failures. As with client-defined operations, standard operations can be invoked directly by either client objects or the Alpha system itself. Exceptions (or other kernel-generated events) may involve manipulating objects asynchronously, in a potentially unsolicited manner, by invoking standard operations on objects.

It is possible for clients to customize an object's standard operations by specifying operations with the same name as system-defined operations. In such cases, the client-provided operations take precedence over the system-defined (default) operations. However the system-defined operations can still be called by their client-defined counterparts (either before, during, or after the client-provided code). This feature proves useful, for example, in supporting compound transactions (see Section 4.3), where the client can increase the performance of applications which perform atomic transactions on objects, by providing a specialized set of operations to replace the standard operations for atomic transactions. The client-defined replacement operations take advantage of an understanding of the semantics of the object's operations to provide the desired effects of atomic transactions more efficiently than would be done automatically by the kernel. This is because there is frequently a way in which the consistency of an object can be achieved following a transaction abort, without restoring each bit of the object to a previously consistent state.

See the Alpha kernel interface specification [Northcutt 88d] for a complete listing of the standard operations defined on objects.

2.1.2.3 Object Attributes

The kernel provides mechanisms to allow clients to construct objects with a range of characteristics that allow differing trade-offs to be made among such attributes as performance and robustness. A typical trade-off makes an object appear more reliable (in the sense that there a greater probability that the effects of operations invoked on it will persist across failures) at the cost of performance (in the sense that the operations will take longer to complete).

When an instance of a specified object type is created, its optional attributes are specified along with its initial parameters. In addition to being able to specify these object attributes when an object is created, the kernel provides a means by which an object's attributes can be modified during the course of an its execution (i.e., via a standard operation that modifies the attributes of the object).

An object's attributes are organized as follows:

- **Permanence:** the attribute of an object that dictates whether an object's state persists across node failures. The options are:

  transient† — the state of the object is lost when a node failure occurs.

  permanent — the state of the object persists across node failures. In this case, the state of an object may be moved between primary memory and its non-volatile image in secondary storage.

- **Atomicity:** the attribute that determines whether an object's permanent state is updated atomically with respect to node failures. The options are:

  non-atomically updated† — the system makes no attempt to ensure that changes to an object's secondary storage image are made atomically with respect to node failures.

  atomically updated — changes to the secondary storage image of an object are made atomically with respect to both normal system behavior and failures. Such an object exhibits the property of atomically changing from one consistent state to another (where consistency is defined by the client on a per-object basis), and at no time can the object be observed in some intermediate state.

- **Availability:** the attribute that governs the degree to which an object remains available for access in the face of node failures. This feature is achieved through the physical redundancy provided by the replication of objects. The options are:

  non-replicated† — a single copy of the object's state is maintained within the system.

  replicated — more than one copy of the object's state is maintained within the system. The replication policy that is used to manage the copies is defined by layers of the system above the kernel.

2.1.3 Key Characteristics

Objects in Alpha may exist only at a single node at a time; however, objects may be dynamically migrated, in their entirety, between nodes. From the kernel's programming perspective, objects exist in a flat universe—i.e., objects are undistinguished by the operating system. Any structure, organization, or discrimination among objects (such as "parent/child" or "system/application") is imposed by the programmer, and enforced by the kernel. Objects are defined by the programmer-in-the-small, as well as the programmer-in-the-large.

In keeping with the goal of providing a simple, uniform programming model, files do not exist as separate abstractions in Alpha; the functionality of traditional files is subsumed by objects in Alpha and so there is no need to introduce another abstraction at the kernel level (although a traditional file system could be constructed above the Alpha kernel). This uniform system interface concept allows operations to be invoked on a wide range of entities, ranging from user or system objects and threads, kernel routines, and system hardware.

Objects in Alpha are similar in many ways to *packages* in Ada [Ada 83]. The main differences between the two are that each object in Alpha exists in a private address space and objects can exist on separate nodes allowing concurrent execution, whereas Ada packages assume shared memory and restrict concurrent access to packages (preferring instead the use of its tasking model).

The Alpha kernel does not take the object model as far as more "pure" object-oriented systems. For instance, in order to perform most efficiently, objects should be medium to large in size—i.e., much larger than integers, and larger than simple procedures, but smaller than entire programs. For example, typical objects in Alpha might contain on the order of 100-10,000 lines of code. This assumption derives from some practical considerations having to do with the distributed nature of Alpha and the overhead associated with both inter- and intra-node

†*These options are the system's default attributes for objects.*

communications. That is, in order for the system to be practical, the overhead associated with locating and accessing an object operation must be a small percentage of the cost of actually performing the function associated with the operation.

Unlike systems such as Smalltalk [Goldberg 83], not all functions in Alpha are implemented as objects; objects in Alpha can be composed of arbitrary programming modules, possessing whatever internal structure is desired (e.g., subroutines). The Alpha kernel reflects the belief that the appropriate granularity of objects is related to the cost of inter-object communication, and that the cost of interprocessor communication in a distributed computer system suggests the use of medium- to large-scale objects. Thus, Alpha supports medium-sized objects that are implemented with more-or-less standard process-style techniques, on more-or-less standard hardware—not on more exotic hardware like capability-based addressing architectures (e.g., CAP [Wilkes 79] and the Intel 432/ iMAX [Kahn 81]).

Any notion of inheritance is assumed to be handled at compile-time; there are no specific features of the kernel's interface that are meant to support inheritance. The standard operations on objects are the only features that could be considered to be (a limited form of) kernel support for inheritance. The kernel provides the code for the default standard operations for all objects in the system, which may be overridden and accessed by client-defined versions of these operations.

The globally unique identifier associated with each object is the exclusive means by which operations can be invoked on an object. In order to control access to objects, the kernel provides a simple capability mechanism [Fabry 74], with which a range of different access control policies can be implemented. Access to objects is controlled by the use of capabilities that encapsulate an object identifier and grant the possessor the right to invoke a given set of operations on the specified object. Since all object interactions are performed through the kernel's invocation mechanism, the capability mechanism provides a global, uniform means of controlling access to all objects. The capability mechanism is described in greater detail in the following chapter.

2.2 Threads

Threads represent loci of execution control that move through objects via operation invocations. Intuitively, a thread corresponds to the conventional notion of a *process*. Unlike conventional processes however, threads move among objects without regard for the physical node boundaries of the system. Threads are the run-time manifestations of concurrent computations in the system—they are the units of activity, concurrency, and schedulability in Alpha. All activity in the Alpha system is provided by threads; objects are passive and all operations are invoked on objects by threads.

2.2.1 Basic Definitions

Threads are independent of any specific object, providing animation for these otherwise passive entities, but having no special association with any one. When, in the course of a thread's execution within an object, an invocation command is encountered, the thread continues execution within the specified operation in the target object. Threads move among objects via invocations independently of the physical location of the objects involved. From a conceptual standpoint, the Alpha notions of thread and object represent the cleaving of the point of execution from the code and data definitions of the standard process abstraction.†

Threads are the form taken by all activity in Alpha—a thread is created each time a new concurrent activity is to be performed. Each thread in the system executes in asynchronous concurrency with respect to the other threads. The system does not limit the number of threads that can be actively executing within an object at any given time. Furthermore, while the Alpha kernel provides mechanisms with which threads can be synchronized, the system does not impose a specific synchronization policy on threads.

Threads can be dynamically created and deleted in the course of an application's execution. When a thread is created, an object and an operation within that object are specified as the initial starting point for the new thread. The initial object in which a thread begins is known as the *root* of a thread (or a *root object*). As a thread moves through the root object, making invocations, it enters and exits other objects in a nested fashion (independent of the physical node boundaries in the system).

Each thread includes the local state information for the computation the thread represents. This information includes the parameters passed to an object on its invocation, the thread's private data (i.e., per-thread automatic variables), and the various attributes of the thread. The attributes of the thread are propagated with it as it moves among objects in the application. Threads have attributes related to the nature of the computation they represent. A thread's attributes communicate the application's service requirements to the system (e.g., robustness requirements, timeliness constraints, and relative importance) to facilitate effective management of the system's resources. The local storage associated with a thread is maintained as a heap—i.e., arbitrarily sized blocks of memory can be allocated and deallocated, either automatically by the invocation facility, or at the explicit request of the client.

Figure 5 is a snapshot of an application running on Alpha, consisting of three separate threads in the process of moving through three different objects in the course of performing their computations. Note that physical node boundaries do not appear in this, the programmer's, logical

---

†*It is noteworthy that this cleaving of activity from static code/data is true at the implementation level as well; in Alpha, a thread together with an object is implemented in much the same fashion as a typical process.*

view. Also, notice that both **Thread$_B$** and **Thread$_C$** are simultaneously active in ***Object$_3$***.

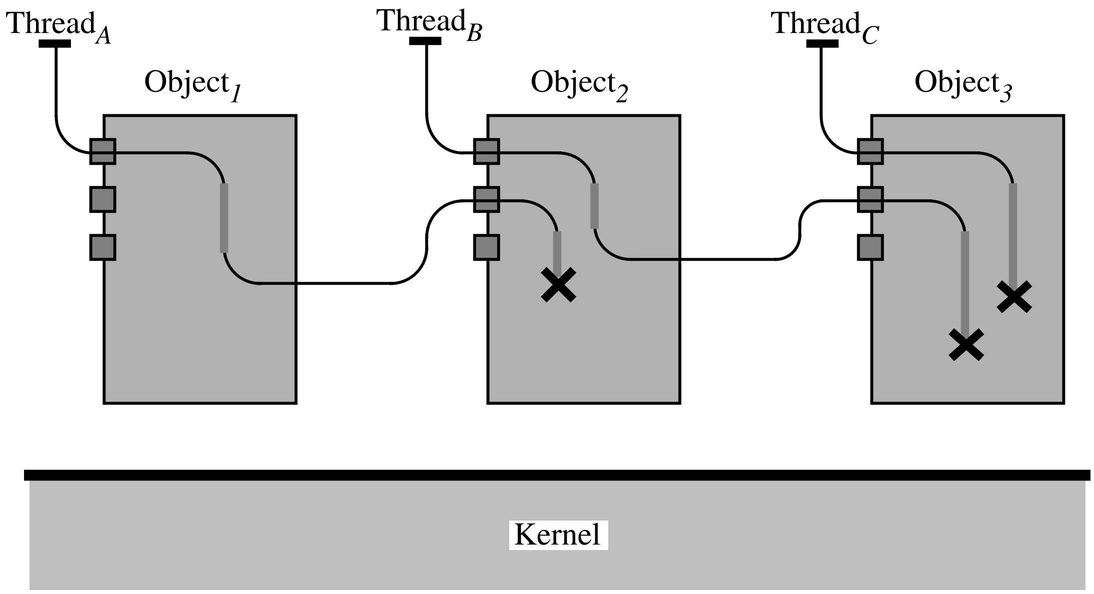

Figure 5: Example Thread/Object Snapshot

2.2.2 Associated Features

As is the case with objects in Alpha, the kernel provides mechanisms for the management of threads, a set of standard operations are defined on threads, and threads have attributes associated with them.

2.2.2.1 Thread Management

Threads are created and destroyed dynamically by invoking operations on the kernel-provided thread management object. To create a thread, a CREATE operation is invoked on the thread management object. Identifiers for the object and operation in which the thread is to begin execution are given as parameters to the invocation of the thread creation operation. The initial attributes to be taken on by the newly created thread (e.g., importance and a time constraint) are also given as parameters to the creation operation. In addition, any parameters for the thread's initial operation invocation are also provided to the thread creation operation.

An identifier for the new thread is returned as a result of a successful thread creation operation. This identifier is used to manipulate the thread, by means of the system-defined operations on threads. As with objects, a thread is destroyed by invoking an operation on the thread manager object, passing the desired thread's identifier as a parameter to the invocation. (See [Northcutt 88d] for a detailed description of the system-provided thread management object.)

2.2.2.2 Standard Operations

In the same way that operations are invoked on objects, operations can be invoked on threads as well. As with objects, operations are invoked on threads with the target thread's identifier as the invocation's destination; invocation parameters may be included as well.

The system-defined operations on threads provide the ability to:

- stop and restart (either immediately or following a delay period) the logical progress of a thread

- abort a thread's current stream of execution
- modify a thread's attributes in a non-block-structured fashion (e.g., change its importance)
- modify a thread's attributes in a block-structured fashion (e.g., begin or end a thread's exception block, time constraint block, or transaction block)
- allocate and deallocate heap storage for a thread's local variables†

Unlike standard operations on objects, the standard operations on threads cannot be customized or overridden by the programmer. Also, there is no means for defining arbitrary operations on threads; the standard operations are provided on threads as a means of manipulating a thread's representation in a fashion that is consistent with the Alpha programming model.

2.2.2.3 Thread Attributes

A thread has associated with it information related to the nature of the computation that the thread represents. This information, which is a collection of the thread's attributes, is used to express application-specific requirements to the operating system at run-time, allowing the system to more effectively manage its resources. The information represented in a thread's attributes includes the computation's reliability requirements, timeliness constraints, and relative importance. This information provides the basis for resolving contention for system resources (e.g., processor cycles, communication bandwidth, buffers, or I/O devices) on the basis of the global characteristics of the individual computations.

In Alpha, each thread represents a client-level computation that has a direct physical manifestation within the kernel. Prioritization of computations can be performed on a per-thread (and therefore a per-computation) basis, as opposed to the per-process basis that is typical in most process/message systems. This is analogous to having a process dynamically alter its priority based on the priority of the process which is the source of the last message received. The direct association of logical computations in Alpha and system entities (and their attributes) helps the system in making effective global resource management decisions. This point is expanded upon in Section 4.1.

Threads carry their attributes along with them as they move through objects (and, transparently, between nodes) in the system. A thread's attributes are modified as it moves through objects. The modification of a thread's attributes is typically performed in a nested fashion. This is represented by the thread in Figure 6, which acquires various attributes in the course of its execution. Figure 6a is a schematic trace of **Thread$_A$**, which began execution in ***Object$_1$***, acquiring its initial attributes on creation, and then taking on an additional set of attributes in the course of its execution within this object. While operating with these attributes, **Thread$_A$** invokes an operation on ***Object$_2$***, and soon thereafter acquires new attributes, then invokes an operation on ***Object$_3$***. Within ***Object$_3$***, the thread takes on another set of attributes, executes with these attributes, discards them, completes the operation, and then returns to ***Object$_2$***. The thread continues executing within ***Object$_2$***, discards the attributes it acquired within this object, and continues execution to the point where the "snapshot" is taken. Figure 6b provides a "straightened-out" representation of the thread, and shows how its attributes change in the

†*This reflects changes made to the original programming model as a part of the Alpha Release 2.0 work being done at Kendall Square Research.*

course of the thread's execution.

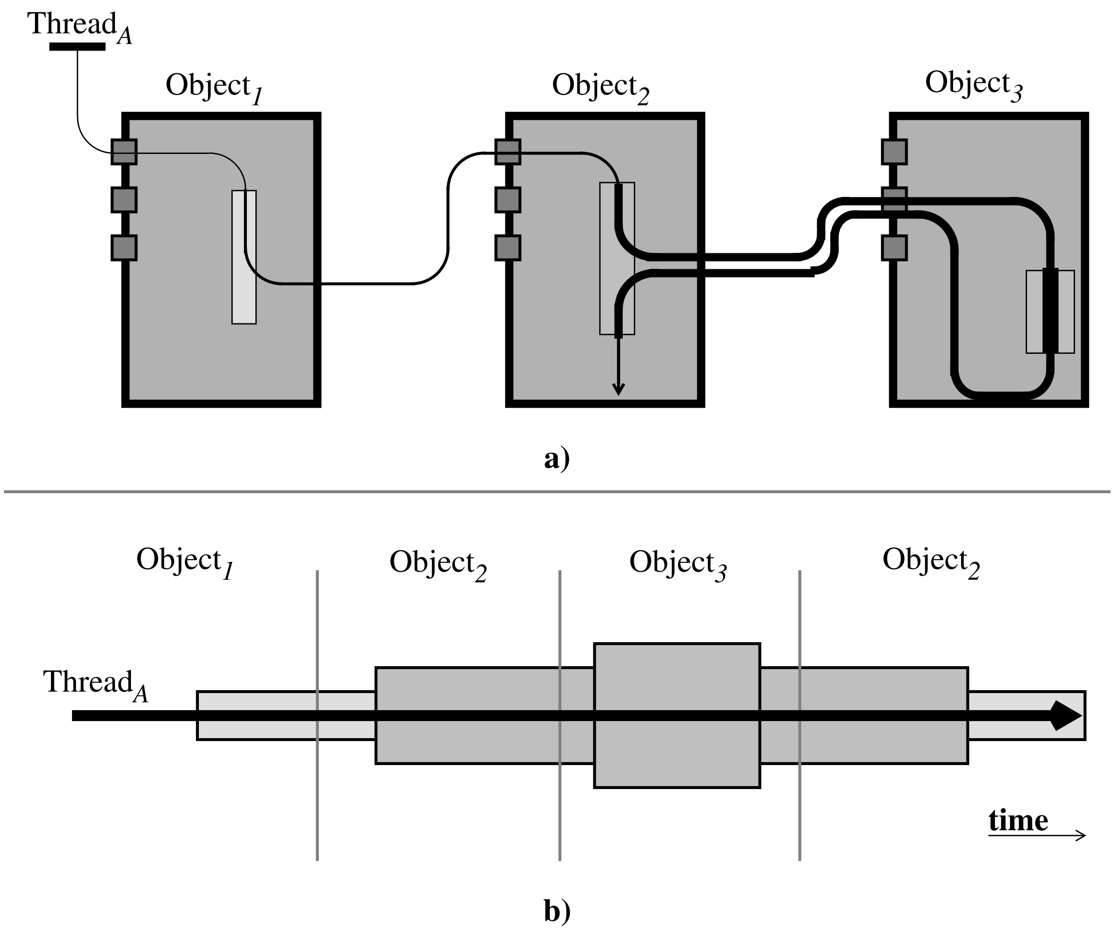

Figure 6: Example of Thread Attribute Nesting

2.2.3 Key Characteristics

The choice of the Alpha thread and object abstractions stems directly from the system's requirements, and there are a number of significant characteristics of the Alpha thread abstraction that set it apart from existing programming paradigms. In particular, with threads it is possible to implement a wide range of system-level control policies, ranging from low-concurrency structures (such as *monitors* [Hoare 74]) to medium- and high-concurrency ones. The thread abstraction simplifies the task of time management in the kernel by being a run-time manifestation of client-defined computations (see the following chapter for further details). Threads are also more efficient than most process- and message-based client/server model implementations, because each step in the computation does not necessarily involve an interaction with the system scheduler (this point will be expanded on in Chapter 5).

The thread abstraction masks from the client issues related to the physical location of objects and the failure of nodes in the underlying distributed system. Furthermore, threads provide a means of exploiting the concurrency inherent in applications, within a distributed computer context, with either uniprocessor or multiprocessor nodes.

2.2.3.1 Orphan Thread Sections

To simplify both the programming model and the implementation, threads in Alpha cannot be forked (i.e., the divergence of a single thread into multiple points of control). However, the same objectives can be met by Alpha since new threads can be dynamically created when a concurrent activity is to begin. While the forking of threads is not permitted under this model, certain types of system failures can generate similar effects. For example, the failure of nodes or communication links can cause threads to become segmented, resulting in what are known as *orphaned computations* [Nelson 81]. Orphans can occur whenever the execution of a thread extends across multiple nodes. A thread is considered to have been *broken* when a failure occurs at a node that lies along the path between the thread's root node, and the node where the head of the thread is currently executing. The sections of a broken thread, other than the one that contains the thread's root object, are known as *orphans*.

Orphans pose a number of problems. First of all, orphans result in an effect similar to the splitting of threads, which is not permitted in the current implementation of the kernel. Furthermore, orphans continue to consume resources as they execute, yet their execution is disconnected from the root of the computation that a thread represents, and therefore cannot contribute to the successful completion of the desired computation. It is therefore important that orphans be detected and eliminated in timely fashion by the kernel. To this end, the kernel's invocation facility supports the notion of *thread repair*. Thread repair involves the detection of the segmentation of a thread, the abortion of all the thread's segments other than that containing the root, and the restoration of the head to the point on the remaining section that is farthest away from the root.

2.2.3.2 Thread Concurrency

Threads are the units of activity, concurrency, and schedulability in Alpha. They execute asynchronously with respect to each other; multiple threads can be active within a single object at any point in time; and multiple simultaneous invocations of operations on an object are possible. Because of this, it is sometimes necessary to synchronize the execution of threads in order to maintain the consistency of the data shared by threads within objects.

Concurrency control could be provided by either the system, which could apply a blanket policy that would restrict the concurrency of all application threads in a similar (brute-force) fashion, or by the application programmer who could be responsible for managing the concurrent execution of the application threads on a case-by-case (i.e., object-by-object) fashion. In Alpha, support for synchronization among threads is provided by a set of kernel-level concurrency control mechanisms (details of these mechanisms will be provided in the next chapter) that allow the necessary kinds and degree of concurrency control to be applied to computations at a cost that is reasonable in terms of overall system performance. The kernel fills its proper role by providing mechanisms for managing concurrency, but does not impose a synchronization policy—specific synchronization policies are defined at the system level in Alpha, or by the application programmer†. Each object is responsible for providing the necessary synchroniza-

†*It is worth noting that the kernel's client can be a language run-time package, in which case the synchronization policy imposed on the client programmer is given by the language definition, and supported by the kernel mechanisms.*

tion for threads executing within it. This approach ensures that the thread concept extends naturally to the multiprocessor case (i.e., with concurrency within nodes, as well as among them).

By placing the responsibility for synchronization with the object, greater concurrency can be obtained than would be achievable by blanket, system-provided synchronization techniques. This is because the programmer of an object can use knowledge of the semantics of the object's operations to maximize concurrency among threads by synchronizing only at those points where it is necessary for only the length of time necessary. While applying brute-force synchronization techniques (such as those used to create monitor-like structures) to objects would relieve the programmer of the burden of providing synchronization, it would also require the entry of individual threads into objects to be serialized, greatly reducing the potential for concurrency.

A number of benefits are derived from the fact that threads represent individual points of control that move among the objects and provide the services required by the computations, and multiple threads can be active within an object simultaneously. For example, the thread and object abstractions in Alpha do not suffer from the nested monitor problem [Lister 77]. Also, when a thread is blocked within an object, it is only that thread whose progress suspended. In a process/message system, each process has but a single point of control, and should a server process block, no other requests can be serviced. For this reason, the progress of all of the server's clients can be affected if a single client causes the server to block.

2.3 Operation Invocation

The Alpha programming model employs a type of Remote Procedure Call (RPC) for the invocation of operations on objects. The operation invocation mechanism is the fundamental facility on which the remainder of the Alpha kernel is based (this is analogous to the role that the interprocess communication facility plays in Accent [Rashid 81]). The operation invocation abstraction defines a facility that provides simple, uniform access to all objects, whether local or remote and provides reliable RPC-like semantics.

The major functions provided by the operation invocation facility in Alpha are: the physical-location-transparent, per-invocation location of the objects that are targets of invocations; the movement of invocation parameters between separate object protection domains; and the management of the exceptions which can occur in the course of an operation invocation.

2.3.1 Basic Definitions

Operation invocation is the means by which all objects interact, and is the global, uniform interface to all client-defined objects, system-provided services, and physical devices in the system. The invocations of operations on objects in Alpha can be nested, and recursive invocations of operations on objects is permitted.

The kernel's operation invocation abstraction was intended to retain as much of the familiar subroutine-call semantics as possible, despite the distributed nature of the underlying hardware. To this end, the operation invocation facility in Alpha allows the passing of arguments between objects, with value-result semantics. It is possible to pass, as arguments in operation invocations, simple or structured data, as well as system-protected identifiers (i.e., capabilities).

Objects can invoke operations on other objects at any point in the course of their execution. The invocation of an operation transfers execution from the *invoking* object to the *invoked* object, and the only data shared between the invoking object and the invoked object is passed in the parameters of the invocation. Each invocation is concluded by a reply, with which the invoked object can return to the invoking object a similar set of parameters.

All operation invocations require an identifier for the destination object, an identifier for the operation to be performed, and zero or more parameters (that may include identifiers of other objects). Similarly, one or more parameters can be returned from an object following an invocation. The one parameter that is always returned from an invocation is a status indication of whether the invocation has succeeded or failed, along with an indication of the cause of the invocation's failure, if appropriate.

All invocation (request and reply) parameters are passed by value, and are passed via the thread's heap area. Two different types of parameters can be passed—*variables* (which can be simple or structured data) and *capabilities* (which are system-protected object descriptors). Also, for each invocation, the specification of a target object and operation can be computed at run-time (i.e., Alpha supports the delayed binding of objects and operation invocations)†.

In Alpha, operation invocations are made independently of the respective physical locations of the source and destination objects. While information about the target object's physical location is not necessary to invoke operations on objects, it is made available for those functions which require it (e.g., task placement or reconfiguration). The physical-location-transparency made possible by the operation invocation facility facilitates the simple and efficient migration of objects from one node to another in the system.

Figure 7 illustrates the behavior of a thread performing an operation invocation. In this example, **Thread$_A$** begins execution within OPERATION$_x$ of ***Object$_1$***, and executes until it encounters an operation invocation command (as represented in Figure 7a). At this point, **Thread$_A$** traps into the kernel, the kernel substitutes ***Object$_2$*** for ***Object$_1$*** in the thread's address space, the kernel (logically) transfers the invocation parameters into the target object's space, and then the thread continues execution within OPERATION$_z$ of ***Object$_2$*** (shown in Figure 7b). When OPERATION$_z$ has completed, the thread traps back to the kernel, which restores ***Object$_1$*** to the thread's address space, (logically) transfers the return parameters to ***Object$_1$***'s address space, and then allows **Thread$_A$** to resume execution following the invocation in ***Object$_1$*** (as illustrated by Figure 7c).

2.3.2 Associated Features

Among the more significant features associated with the operation invocation abstraction in Alpha are the application of controls to restrict the ability of individual threads to invoke operations on given objects, the facility's ability to manage the exception conditions which may occur when invoking operations, and the manner in which a thread's flow of control is constrained by the operation invocation abstraction.

2.3.2.1 Fault Containment

The reliability requirements for Alpha (as defined in [Northcutt 88a]) indicate the need to provide a high degree of fault containment in the system. Fault containment involves the attempt to limit the effects of the failure of one component on the other components within a system [Levin 77, Boebert 78]. Ideally, fault containment would guarantee that a failed object

---

†*This represents a change made to the original programming model as a part of the Alpha Release 2.0 work being done at Kendall Square Research.*

could not interfere with the operation of any other object. The Alpha kernel's access control mechanisms are designed, however, to only limit the scope of interactions each object may have, and thereby limit the potential extent of a failed object's damage.

The primary means of supporting fault containment in Alpha is by way of system-provided and enforced *protection domains*. The kernel places each object in a separate address domain, enforces this separation with the underlying hardware, and controls all interaction among objects and their domains. To provide the desired degree of protection, the access control mechanisms in Alpha ensure that each object can invoke operations on only those objects for which it has explicit permission to do so. By enforcing the separation of object protection domains, the general system objective of fault containment is advanced.

The control of access among protection domains in Alpha is provided by a *capability* mechanism. The invocation of operations on objects is controlled by the kernel through the use of a system-protected identifier (i.e., a capability). In this way, the ability of objects to invoke operations on other objects can be restricted to only that set of destination objects explicitly permitted. A capability is an object's local manifestation of a system-protected object identifier, and provides the object with a means of accessing the designated object.

To invoke an operation on an object in Alpha, the invoking object must possess a capability for the target object. The very fact that an object possesses a capability implies that the object has the right to access the object referenced by the capability (subject to any restrictions associated with the individual capability). Since all interactions among objects in Alpha are via invocations and all invocations use system protected names, capabilities provide globally uniform access control. Capabilities are used to uniformly solve both the problems of object addressing and object access control in Alpha. Details of the Alpha access control mechanisms are provided in Section 3.1.

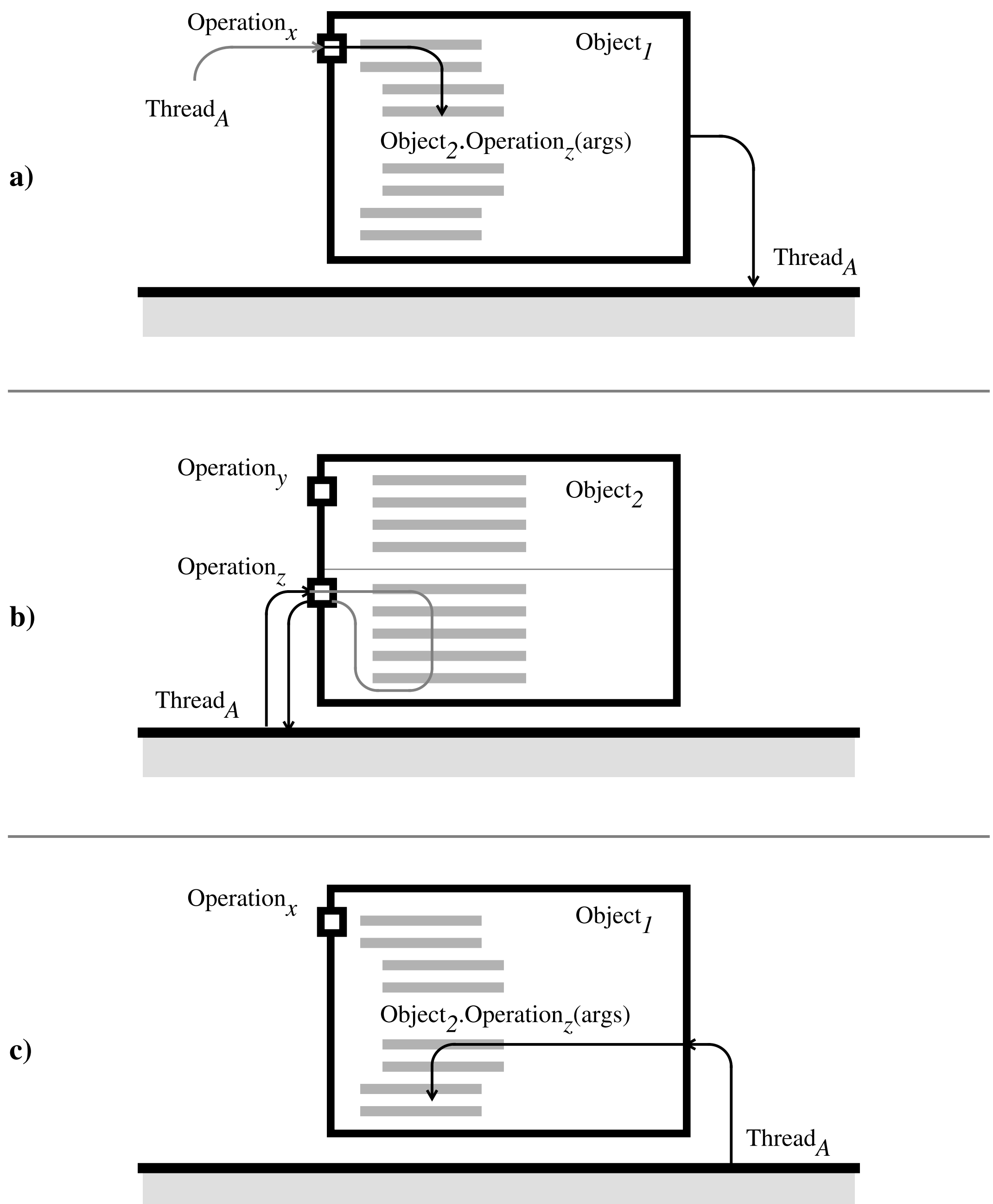

Figure 7: Example Operation Invocation

2.3.2.2 Invocation Exceptions

The operation invocation facility does much to mask the undesirable effects of the system's physical distribution (e.g., node failure, packet/message errors, non-local objects, and object migration) and provides time-driven orphan detection and elimination. This is to say that the operation invocation facility makes every attempt to guarantee that the desired operation invocation has been completed (once, and only once), and if the system cannot be certain of the successful completion of an invocation, the client is notified and all orphaned thread fragments are destroyed. Details of the exception handing mechanisms for the operation invocation facility are given in the following chapter.

Each invocation returns a success/failure indication on its completion. The kernel provides an error code along with the operation invocation failure indication, to notify the client that an exception has occurred during the invocation. The types of invocation exceptions that are detected by the kernel include:

- an invalid target object capability
- target object not found
- invalid or undefined operation
- bad capability passed to/from the target object
- thread break occurred during the invocation

(See [Northcutt 88d] for a complete description of the system's invocation failure status indication values.)

It should also be noted that failures which occur above the kernel level (i.e., within the target object's operations) are indicated via the normal return parameters for the operation—these errors occur at a higher level of abstraction within the system and therefore are handled at a correspondingly higher level in the system.

2.3.2.3 Flow of Control

Operation invocation serves as both the means of interaction among objects and the mechanization of the interface between an object and the system. These are clearly at two different levels of abstraction; the interaction between the invoking object and the system is at the lower level, while the interaction between the invoker and the invoked entity is at a higher level. At the lower level of abstraction, the control behavior is by nature synchronous—when the invocation is made the invoking entity's logical progress is suspended while the system (and possibly the invoked entity) performs work on the invoker's behalf. Alternatively, the type of control behavior found at the higher level of abstraction can generally be categorized as either *synchronous* or *asynchronous*. Synchronous behavior at this level is represented by remote procedure calls, where the invoking process is suspended until the invoked process completes the operation specified in the invocation. At the same level, asynchronous behavior is represented by message-passing systems where the process which sends a message is suspended only to the point where the system can register the message transmission request and then the invoker's progress continues, independent of the destination process.

The lower level of communication must be considered separately from the higher level; the lower level is by nature synchronous, while the higher level can be either synchronous or asynchronous. Since the lower-level invocation semantics must be synchronous, the only interesting design choices deal with the time at which the invoking entity may resume execution (i.e., the time at which the invocation returns). Examples of possible return times and the meanings associated with them include: "the invocation request has been noted", "the invocation request is in the process of being serviced", "the invocation message has been delivered to the destination object", "the invocation message has been acknowledged by the destination object", and "the invocation has been performed by the destination object and has responded." The higher-level aspect of communication semantics is most commonly thought to be a binary choice between synchronous and asynchronous communication services. In Alpha however, we note the continuum of choices available for an invocation mechanism, and choose the semantics most appropriate for our system. Therefore, the object invocation mechanism in Alpha is an extremely synchronous type of high-level, end-to-end communication.

The Alpha kernel does not provide a message-style communication facility, and there are currently no plans for adding one. This is not a limitation since it is believed that the Alpha programming model is highly expressive and permits such a high degree of concurrency that an asynchronous message-passing facility is not called for. Nonetheless, should a communications facility with asynchronous message-passing semantics be desired, it is possible to construct such a facility with the existing mechanisms. For example, a ***Port*** object type might be defined and instances of this type can be created. A SEND operation on the port object could queue a client-defined *message* on the port. Such a message could then be received by invoking the RECEIVE operation on the ***Port*** object, which would pass back a message among its return parameters. Such a ***Port*** object could be a client- or system-defined object, and could be made as efficient as a native message-passing implementation.

2.3.3 Key Characteristics

The form of thread/object interaction provided by operation invocation has a number of desirable features. The RPC style of communication used in Alpha tends to be both simpler and more commonly understood by programmers than more general forms of communication (e.g., asynchronous message-passing) [Nelson 81]. Due to the nature of the object and thread abstractions in Alpha, the synchronous form of communication provided by the invocation facility does not suffer the limitations typically associated with synchronous communication in process/message-based systems [Liskov 85]. Furthermore, it is believed that this programming abstraction can be implemented efficiently enough to permit the construction of meaningful applications on top of the system.

The fact that objects exist in separate, hardware-enforced address spaces does not imply that the cost of a full context swap is incurred as a result of each operation invocation. The only valid implication is that the memory translation tables must be altered on invocation, to replace the current object table entries with those of the invoked object. This activity incurs a much lower cost than a true context swap—where processor state and potentially all of the memory translation table entries must be saved and reloaded. Details of how the operation invocation facility is implemented in Alpha are provided in [Northcutt 88c].

Because of the fact that all of Alpha's system services masquerade as objects, the operation invocation facility subsumes the role of traditional "system calls" and constitutes the single entry point into the system. Thus, the operation invocation facility provides a uniform, location-transparent means for accessing all system- and client-provided services. The fact that invocation is the only manner in which objects can directly interact has a number of important benefits. Invocations provide the kernel with complete visibility of interactions among objects. The kernel can track the movement of threads—a feature that is useful in supporting atomic transactions, monitoring, and debugging. No alternative channels of communication exist, so the kernel can create an accurate model of the interactions among objects, and can follow the execution of computations through successive invocations of objects—all of which contributes to the system's ability to manage system resources more effectively. The invocation mechanism serves as the single point through which all data is passed among objects. This provides an obvious point where translations can be performed on exchanged data to accommodate different machine-dependent data representations.

In the current research implementation of Alpha, a number of programming details concerning operation invocation depend largely on the choice of the language interface provided to the kernel's client. Included among these details are: the actual syntax of operation invocations; the manner in which the programmer specifies the parameters to be passed and returned; how the programmer names object types and instances; the way in which capabilities appear to the programmer; and how capabilities are distributed among objects. It is considered the responsibility of this language to manage the initial access restrictions by distributing the initial capabilities to objects and to perform whatever degree of compile-time analysis is desired (e.g., invocation parameter type-checking). The kernel provides a powerful set of mechanisms to support such a language interface, but it is the responsibility of the language designer to decide exactly how these mechanisms are to be presented to the client. Since this research project does not enjoy the benefit of an accompanying compiler effort, object programming has been performed in the C programming language, with a set of simple extensions provided by a pre-processor. Appendix I provides an illustration of some example code written for use with the Alpha kernel, and [Shipman 88] provides a description of the language extensions.

3 Ancillary Programming Abstractions

This chapter describes a set of abstractions that the system provides to augment its basic abstractions. Described here are the access control abstractions that are used in conjunction with operation invocation, and the concurrency control abstractions that are provided in support of objects and threads.

3.1 Access Control Abstractions†

As outlined earlier, objects in the Alpha system are placed in separate address spaces, and all access to objects is controlled by *capabilities*. In order to invoke an operation on an object, the invoker must own a capability to the desired target object, and the capability must permit the invocation of the desired operation.

The kernel supports capabilities as a basic system abstraction, and provides mechanisms that allow desired applications-level access control policies to be implemented through the distribution of capabilities among objects.

3.1.1 Basic Definitions

A capability is analogous to a logical pointer to an object that cannot be forged, nor directly manipulated by objects. Capabilities in Alpha are long-lived, i.e., they exist independently of the lifetime of the objects that create them and of the objects they represent. Capabilities are context independent—i.e., they refer to the same object regardless of their current domain. When a named entity in Alpha (e.g., an object or a thread) is instantiated, a capability for the newly created entity is returned, that can subsequently be passed as an operation invocation parameter.

Each object in the system has an associated collection of capabilities (known as a *c-list*) that defines the object's current access domain (i.e., the other objects on which it can currently invoke operations). Capabilities are added to an object's c-list in only two ways—by having them passed to the object as invocation parameters that are explicitly copied into the object's c-list; or by having them inserted into the object when it is instantiated (i.e., passed as a part of the object's initialization parameters).

When a capability is passed into an object as an operation invocation parameter, the capability may only be accessed by the thread making the invocation. Hence, the invoking thread can use its passed capabilities in addition to those that appear in the invoked object's c-list, while, other threads in the invoked object are not aware of these passed capabilities. When the invocation is complete (i.e., when the invoking thread leaves the target object), the capabilities passed into the object by that thread are removed. Passed capabilities are managed like the automatic parameters of a procedure call in a typical block structured programming language—i.e., they are added and removed from the object's access domain as nested invocations are initiated and completed, much as stack frames are allocated and deallocated as nested calls are made and completed. In addition, capabilities passed back as reply parameters are brought into the invoking object with the returning thread, and the visibility of these capabilities is restricted to the invoking thread alone.

Capabilities are added to and removed from an object's c-list as a result of an explicit action taken by the thread that owns the capability that is an invocation parameter for that thread. In

---

†*This discussion reflects changes made to the original programming model as a part of the Alpha Release 2.0 work being done at Kendall Square Research.*

order to make a capability visible to other threads within an object (and to persist beyond the invocation in which it was passed), the thread that has access to a passed capability must perform an explicit operation on the object to add the capability to the object's c-list. A capability can be explicitly destroyed (i.e., removed from an object's c-list) only by the object that owns it. Note that capabilities that are not in an object's c-list can be lost when the threads that are carrying them are destroyed (e.g., due to thread breaks). Otherwise the capabilities in an object's c-list remain in existence along with the object, independent of any particular thread or other object in the system.

It is the responsibility of the creator of an object to ensure that the necessary capabilities are provided to the newly instantiated object. The capabilities to be passed, either into or out of the object, are specified as part of the object's interface (within the formal parameter list of the given operation), and appear in the proper location in the actual parameter list of an invocation. If a capability that is to be used as an actual invocation parameter contains restrictions that prohibit it from being passed to other objects, then the invocation will fail and the system will indicate that the capability was restricted from being passed.

The kernel maintains the actual representations of all of the capabilities in the system and employs the hardware's protection mechanisms to limit access to the internal representation of capabilities. Conceptually, the kernel-maintained representation of a capability consists of a globally unique object identifier used to address an object (or other system entity), and a per-operation set of usage restrictions, that dictate how the capability can be used by the object that owns it.

Each time an object is invoked, all of the capabilities used in the invocation are validated. Checking the validity of a capability involves determining if the specified capability is owned by the object from which the invocation request or reply is being made, and if so, verifying that no restrictions associated with the capability prevent it from being used in the indicated fashion. Then, as part of the invocation procedure, the kernel checks that the capability for the invoked object is valid, and similarly checks the validity of all capabilities passed as invocation parameters.

3.1.2 Associated Features

When an object is created, the capability returned by the kernel-level object manager is unrestricted—i.e., the capability can be used to invoke any operation defined on the newly instantiated object, and the capability can be passed as a parameter in operation invocations. The restrictions that can be placed on individual capabilities can limit their ability to be used to invoke particular operations on the object to which they refer, and constrain the manner in which they can be passed to other objects. Objects can further restrict capabilities, but they cannot remove restrictions that have already been applied to capabilities. That is, the kernel provides a *restrict* primitive, but no primitive for rights amplification. It should be noted that, while the kernel's object manager always returns unrestricted capabilities for newly instantiated objects, a system-level object manager could be created that uses the kernel-level object manager, but also applies restrictions to the capabilities for the newly instantiated objects that are returned to users. In this way, the system can keep clients from obtaining access to an object's system operations, and can be made to refuse to create certain kinds of objects for a given client (e.g., as a result of an access-list or user-authentication activity).

The Alpha kernel provides a set of standard (system-defined) operations on all object types that permit the manipulation of capabilities. These operations allow restrictions to be placed on capabilities, allow the capabilities passed to an object via invocations to be added to an object's c-list, and makes it possible to both duplicate and destroy capabilities within an object's c-list. In order to manipulate a capability, an operation is invoked on the object which contains the desired capability within its c-list, passing the (object-local) identifier for the capability as a parameter to the operation. Some of the capability manipulation operations (e.g., the COPY operation) cause a new capability to be added to an object's c-list, while other operations (e.g., the RESTRICT operation) modify the state of a given capability, and yet other operations (e.g., the DESTROY operation) remove a capability from an object's c-list.

The capability mechanism in Alpha has special features to support the concept of *Well-Known Objects (WKOs)*. WKOs are simply objects that have "well-known" names— i.e., they have identifiers that are known at application configuration time, as well as at run-time. Capabilities for WKOs are known as *Well-Known Capabilities (WKCs)* and are simply the capabilities that are used in the invocation of operations on WKOs. A WKO will have at least two different internal system identifiers—one will be the unique identifier assigned at run-time when it is created; and the other is the well-known name that it is assigned when the application is configured (and is used by the kernel as an alias for the WKO). A WKC for any object is entirely equivalent to the same capability referring to the WKO by its run-time kernel identifier, rather than its well-known identifier.

Well-known capabilities act only as aliases, and are not intended to provide any more functionality than ordinary capabilities. In particular, objects that are invoked by means of WKCs are not necessarily more robust than other objects. If the WKO is to be robust, it must use the Alpha-supplied robustness mechanisms to satisfy its reliability and survivability requirements. Notice, however, that a WKC could be used to allow invokers of a WKO to use a single identifier to reference the WKO—even if several different objects actually provided the service or data over a period of time. (Of course, the regeneration of these object instances is to be handled by the robustness mechanisms.)

Well-known capabilities address two concerns:

1. WKCs eliminate a start-up circularity arising from the fact that an object must hold a capability in order to perform an operation invocation. The capability used to make the first invocation in the system must either (a) be received as a parameter in an incoming invocation, (b) be received as a return parameter from a previous outgoing invocation, or (c) be held by the object since its creation. The first two cases are impossible because they involve invocations that must occur before the first invocation in the system. Therefore, only the third case is possible, implying that at least one object must possess at least one capability at the time of its creation in order for any invocations to be performed. WKCs provide the method for declaring and supplying such capabilities.
2. WKCs permit the definition of static application configurations, thereby allowing Alpha applications to be created in a fashion more similar to traditional real-time applications. Since Alpha represents a radical departure from typical real-time supervisory control systems, it is often advantageous to design and implement Alpha's features and facilities so that they can be related to (and be at least as efficient as) the features and facilities found in more traditional systems. (For instance, although Alpha provides the functionality of a secondary storage system and offers a dynamic paging facility, there is no requirement that these be used; rather, the user

may choose to “wire down” all of the objects in the system into a sufficiently large primary memory.) In the case of capability distribution, Alpha accommodates a range of approaches. On the one hand, a very dynamic approach uses a single object that initially creates all of the other objects and threads in the system at start-up time. This “master” object can then invoke each of these newly created objects in order to supply them with the capabilities that they will need. On the other hand, Alpha also supports a totally static system—one in which every object is "hardwired" to all of the other objects with which it must deal. In this case, when each object is created, it is given all of the capabilities that it will ever need. This approach is facilitated by WKCs—each object would use WKCs to “know” about all of the other objects it would ever access.

A mechanism to register WKOs is offered by the kernel, and is used by objects that need to assume a well-known identity. In fact, this mechanism is used at initialization time by the kernel to register the kernel-provided objects under common, well-known aliases. Currently, the kernel also registers the object types of which it is aware. (In the future, this mechanism may also be used to register the available object type definitions that are stored in the storage management subsystem). Additionally, the mechanisms that locate and manipulate objects based on capabilities handle WKCs as aliases that are equivalent to capabilities using the objects’ unique kernel identifiers.

All WKCs are generated by the Alpha configuration program (ACP [Shipman 88]). At configuration time, the configuration manager must assign kernel identifiers to all of the WKOs, including:

- the kernel-provided objects at each node,
- the object type definitions that are referenced in the configuration, and
- the objects that are declared to be WKOs in object definitions (using the interim object programming language constructs [Shipman 88]).

3.1.3 Key Characteristics

The intent of the access control abstractions and underlying mechanisms in Alpha is to provide the system with defensive, not absolute, protection. This approach is taken primarily to limit the scope of this research effort to the issues of more immediate concern and greater research value. Furthermore, the embedded nature of most real-time command and control computer systems restricts (but does not eliminate) the opportunities for mounting determined attacks on the system. The emphasis in this work is more on providing a reasonable degree of assurance (at a moderate cost) that programming errors will not lead to serious system failures. It was decided that a full capability system (one in which a capability is required for virtually every bit accessed) is not called for in Alpha, and a more suitable solution would be a higher-granularity type of protection scheme. The access control mechanisms used in Alpha fit the Alpha object model—i.e., protection is performed on a per-object and per-operation basis, as opposed to a memory segment basis. This choice of protection mechanisms was motivated by the desire to construct the kernel on traditional hardware, the goal that protection should add only a small amount of overhead to the cost of operation invocation, and the belief that there is little is to be gained (in this context) from fine-granularity protection.

The Alpha kernel provides mechanisms that can be used to enforce various protection policies. One policy for capability distribution requires that all objects are created with a single, well-known capability for an object known as a *name server*. The name server object would typically have operations defined on it to allow objects to associate themselves with service names (e.g., client-defined strings), and operations to return certain capabilities to objects associated with various service names. In this way capabilities could be distributed to objects at run-time, with a single name server object providing service for all other objects in the system. Also, higher-level access control schemes (e.g., access control lists) could be applied using the name server object concept. Alternatively, an object could encapsulate a set of capabilities for each object, which would permit the implementation of *environment objects* that provide a standard set of capabilities for objects. Such environment objects can be modified and inherited in much the same way as UNIX shell environment variables.

The protection mechanisms provided by Alpha are more similar in this respect to the protection schemes found in some message-passing systems than those of object- and capability-based systems. The capability mechanism used in Alpha is similar to that found in some message-passing systems where protection is provided by controlling communication among processes (e.g., via port control, such as in [Baskett 77] and [Rashid 81]), and is most closely related to that of the (centralized) CAL system [Lampson 76]. The protection service of Alpha is much less comprehensive and provides a lesser degree of protection at a more modest cost than the protection facilities in such systems as Hydra [Wulf 81], StarOS [Jones 79] and System/38 [Berstis 80].

It should be noted that the Alpha protection abstractions and mechanisms, like all of the abstractions and mechanisms in Alpha, are not necessarily intended to be the primitives ultimately used by the application programmer. Exactly how capabilities appear to the object programmer, and how they are passed in invocations, depends on the specifics of the language or operating system that is to be built on the Alpha kernel. The kernel mechanisms provide the kernel's clients with a means of inter-object protection.

As with many other systems that use capabilities [Levy 84], Alpha does not provide for the actual revocation of capabilities. It may be worth noting however, that automatically deleting the thread dependent capabilities when the thread returns from its invocation on the object represents, in some sense, a limited form of revocation.

In addition, the Alpha access control facility affords a greater degree of fault containment than the direct access (or access path) approach taken in other object-oriented systems that use capabilities more extensively than Alpha [Jones 79, Wulf 81]. In Alpha, protection domains are compartmentalized, in that each object can only invoke operations on objects for which it has a capability. Other systems allow objects to use capabilities that they themselves do not have by referring to capabilities indirectly, through a chain of other objects' capabilities (e.g., *paths* in Hydra). While objects can still fail in Alpha, they can only access (and hence interfere with) those objects for which they themselves have capabilities. In this way, the compartmentalization of object protection domains helps to confine the effects of object failures. A compartmentalized approach to capabilities allows programmers to set up firewalls that can detect the aberrant behavior of a failed object and halt the propagation of its effects. However, in a system where paths can be used in place of simple capabilities (allowing indirect access to objects via chains of capabilities), if an object fails it can directly manipulate objects outside of its immediate protection domain.

3.2 Concurrency Control Abstractions

Threads, as they are defined in Alpha, are unconstrained with respect to their execution relative to one another—in particular, multiple threads can execute concurrently within a single object. In order to construct applications that behave correctly and predictably, it is necessary to provide abstractions for controlling the concurrency among threads. The concurrency control abstractions provided by Alpha allow a client to restrict the concurrent activity of threads where necessary to achieve the desired system behavior, while still meeting the goal of maximizing concurrency of the threads in the system. Another objective of the concurrency control mechanisms in Alpha is to provide support for a range of reliable, modular programming disciplines (e.g., various object-oriented languages and atomic transactions).

Typically, concurrency control mechanisms are based on the notion of controlling (in the sense of starting and stopping) the logical progress of computations. In Alpha, this corresponds to the control of thread execution. Therefore, the control of the virtual progress of threads is the basis for all concurrency control mechanisms in Alpha and is the fundamental technique that supports higher-level synchronization facilities.

Because the object model constrains all access to data to originate from within the object that encapsulates that data, the implementation, verification, and application of the abstractions by the user of various synchronization conventions is greatly simplified. In much the same way that monitors are an improvement over the use of generalized critical sections, objects centralize the location of the code that shares access to particular pieces of data, and therefore also centralize the locations where concurrency control is required. Furthermore, object-oriented program structures make possible the use of semantic information concerning the operations on the constituent data to obtain greater application-level concurrency than is possible through more standard techniques. For example, with objects it becomes more reasonable to consider such measures as enforcing orderings on the individual steps of the operations defined on an object in order to obtain increased concurrency [McKendry 84b].

In Alpha, there are two kernel abstractions that control thread execution—*semaphores* and *locks*. These concurrency control abstractions both provide the means for starting and stopping the logical progress of threads in order to achieve the desired synchronization among computations. However, semaphores and locks accomplish their functions in two different ways: semaphores provide control over the number of threads that can simultaneously execute within a region of code in an object; locks control the number of threads that can perform various (read or write) operations on a given region of an object's data. Together, these mechanisms provide a complementary and orthogonal pair of synchronization abstractions, upon which arbitrary (higher-level) synchronization policies can be implemented (typically at the system level of Alpha).

The kernel-level synchronization abstractions in Alpha were designed and implemented to provide support for the needs of real-time applications. For example, the synchronization primitives are designed to consider timeliness constraints in their function. Each time a thread performs a synchronization operation (i.e., manipulates a semaphore or a lock), the scheduling subsystem is notified. When the time comes to unblock a thread waiting for a *synchronization token* (e.g., when a `V` operation is performed on a semaphore, or when a lock is released), the timeliness attributes of each of the blocked threads is considered in choosing which thread to unblock. Furthermore, the timeliness attributes of all of the threads waiting for a synchronization token held by a thread is considered by the kernel when determining when a thread should be preempted. In some cases, it may be desirable to allow a thread with less stringent timing constraints to execute before one with tighter constraints in order to allow a thread with even tighter time constraints to be unblocked, following the release of a synchronization token†. Behavior such as this is dependent on the system-level scheduling policies that the system implements; the kernel-level concurrency control mechanisms provide only the basic functions and the hooks necessary to allow a wide range of policies to be carried out.

For reasons of adaptability, the attributes of threads (e.g., time constraints) are (by convention), modified in a block-structured manner and are strictly nested within object operations (e.g., a time constraint block begins and ends within the same object operation). This programming convention is encouraged by the system and provides a modular, structured way of managing thread attributes, and is not unlike the way in which monitors confine all of the P and V operations on semaphores into a common module in order to add structure to, and avoid the problems of, distributed synchronization. A similar approach is taken for the use of the kernel's synchronization abstractions—semaphores and locks can be used to control thread concurrency within code and data regions that lie entirely within the scope of a single object.

3.2.1 Semaphores

The first (and most basic) synchronization abstraction provided by the Alpha kernel can be used to provide functionality similar to that of *critical sections*—a construction that restricts the concurrent access to a given region of code to a maximum of N threads within an object. Critical sections are useful for ensuring that at any one time, a maximum of some given number of threads can be executing a section of code within an object. Once the maximum number of threads have entered and are executing within a critical section, all other threads that attempt to enter this part of an object are made to wait (i.e., they are *blocked*) until one or more of the threads leaves the critical section. A special case of critical sections, i.e., where N = 1, can be used to enforce the mutual exclusion of threads within a region of an object. A mechanism commonly used to implement critical sections is known as a *counting semaphore*, which Alpha supports through kernel-provided ***Semaphore*** objects and a semaphore management object.

Semaphore objects feature operations corresponding to the *P* and *V* primitives of traditional semaphores, along with a non-blocking (or conditional) *P* operation (C_P). The C_P operation, like the standard P operation, attempts to acquire a (logical) synchronization token, however, it never blocks the thread as a P operation would if a synchronization token was unavailable, but instead returns a success or failure indication. The P and V operations allow the creation of critical sections within an object's code (which is the system's most fundamental concurrency control construct), and the C_P operation allows the implementation of "spin-lock" services. ***Semaphore*** objects are created and destroyed by invoking CREATE and DELETE operations on the kernel-provided semaphore management object. Details of the interfaces to the semaphore and semaphore manager objects can be found in [Northcutt 88d].

As an implementation-guided optimization, the capability passed back to an object by the semaphore manager as a result of an operation to create a instance of a ***Semaphore*** object is restricted from being passed on to other objects. This restriction ensures that all accesses to a semaphore are made by the object that created them. This restriction is useful in distributed

---

†*This should be contrasted with the FIFO ordering discipline that is typically used to provide fairness (a property that is not consistent with the needs of real-time systems), or to avoid starvation (a condition that may be perfectly acceptable in a real-time system).*

systems that permit the dynamic reconfiguration of objects, because it allows the system to keep each semaphore and the object that uses it on the same node. This association between semaphores and their users is provided because the overhead associated with having multiple objects on different nodes directly access the same ***Semaphore*** object was considered to be prohibitive (and the kernel currently does not offer a mechanism to allow the client to specify that a collection of objects should remain co-located, regardless of the reconfiguration activities carried out by the system or the client). Furthermore, this restriction is not considered a limitation, in that it does not restrict the types of control structures that can be implemented in Alpha.

There are a number of observations to be made on the nature of the usage of semaphores as a thread concurrency control abstraction in Alpha. It is expected that the language (or operating system) that exists on top of the Alpha kernel will eliminate the need for the explicit use of this synchronization abstraction by the client. For example, a language might provide a block-structured *CriticalSection* primitive, a *Monitor* declaration, or some higher-level synchronization primitive (e.g., path expressions[Campbell 74]). In such languages the allocation and deallocation of ***Semaphore*** objects, as well as the insertion of P and V operation invocations at the appropriate points within objects, would be performed automatically. A client-level synchronization policy could allow some of the synchronization activities to be implicit, even if it results in less than optimal concurrency.

The semaphore abstraction can be used in conjunction with objects to implement monitor-like structures. For example, when an object is initialized it could create a ***Semaphore*** object with a count initially equal to one. Each operation in the monitor-like object could then begin by invoking a P operation on the previously allocated ***Semaphore*** object, and end with a V invocation on the same ***Semaphore*** object. This ensures that, like a monitor, each object that adheres to this discipline can have exactly one thread active in it at any time. It should be noted, however, that such monitor-like structures quite severely restrict a computation's concurrency, and much greater concurrency can be achieved through a more judicious use of the Alpha concurrency control abstractions. For example, in many instances, higher concurrency could be achieved if critical sections were used only in those parts of the code where undesirable side-effects may occur due to the concurrent execution of threads within an object. That is, concurrency can be increased by reducing the granularity of synchronization. It is also worth noting that semantic information concerning the operations being performed by an object may be applied in order to achieve greater concurrency; if nothing is known about an object's operations, only simple, conservative forms of synchronization can be used.

An example in which semaphores are used to coordinate the activity of multiple threads is shown in Figure 8. In this example, a semaphore is used to enforce the dual restrictions that: the number of threads active in the REMOVE operation must be less than or equal to the current number of elements in the queue, and the number of threads active in the INSERT operation must be less than or equal to the current number of free entries in the ***Queue*** object.

At the kernel level of Alpha, semaphore-induced thread deadlocks are to be detected and resolved by Alpha's time-driven scheduler. However in the current release of Alpha, deadlocks are dealt with by way of time-outs. Because the kernel already provides a time-out mechanism (in the form of thread time constraints) there is no need to introduce a special mechanism for dealing with deadlocks, or to include a time-out mechanism as part of the ***Semaphore*** object's operations. Instead, the kernel mechanisms allow a hard time-constraint (i.e., deadline) to be placed on the amount of time that a thread is willing to wait while attempting to acquire a synchronization token. Should a thread wait for a semaphore for longer than the specified amount of time, the system will notify the thread, allowing the thread to give up and carry out a recovery procedure.

Conceptually, a semaphore manages logical system resources (known as synchronization tokens) that represent a thread's ability to execute within a region of an object where concurrent access is to be controlled. The kernel must not lose track of these synchronization tokens, or else threads could be blocked indefinitely, or access to certain regions of an objects could be permanently denied to all threads.

In Alpha, the kernel provides mechanisms to permit reliable programs to be constructed at higher levels, without imposing a particular reliability policy. In keeping with this approach, an application is responsible for managing the reliability (i.e., consistency, correctness, and availability) of application resources. ***Semaphore*** objects, however, are system resources and should therefore be managed by the system. In order to ensure that the consistency of the semaphores used by an object is maintained despite the abortion of threads that might have affected its state, the system must receive information from the client concerning the manner in which the semaphore object is to be used.

There are a number of actions that could be taken when a thread that is either waiting for a synchronization token, or is already holding one, is aborted. For example, when a semaphore is used to enforce mutual exclusion on a critical section of code within an object, the system should reclaim the synchronization token from any thread that is aborted while executing the critical section. On the other hand, when semaphores are used to coordinate a producer-consumer relationship among threads as shown in Figure 8, the logical synchronization tokens represent full and empty data item entries in a buffer. Should a producer thread be aborted while holding a synchronization token (representing an empty buffer slot), the token should be returned to the system (to represent the return of the empty slot to the buffer). If a consumer thread acquires a synchronization token (representing a data item from the producer) and then aborts, the token should be returned to the system (to indicate that the produced data has been returned to the buffer, in order to be consumed once again). Note that the application must be carefully written to permit data to be partially, but not completely, processed—and even reprocessed.

The corrective action that must be taken when such a thread is aborted must be specified when a ***Semaphore*** object is created. This is done by way of a parameter that is given to the semaphore manager object when a ***Semaphore*** object is created. Details of this interface is given in [Northcutt 88d].

```
/*
* An Example Queue Object Type Specification
*/
OBJECT Queue() {

/*
* Declarations
*/
#define Q_SIZE          100

static boolean          init = FALSE;
static char             queue[Q_SIZE];
static int              qtail,
                        qhead;
WELLKNOWN CAPA          SemaphoreManager    SemObj;
CAPA                    Semaphore           FullSlots,
                                            EmptySlots;

/*
* Operations
*/
OPERATION Initialize()
        /*
         * Initialize the queue object by allocating the semaphores, setting the initialized flag,
         *  and then returning successfully.
         * Return a failure if the allocation of semaphores is unsuccessful.
         */
        {
                /* allocate and initialize the 'FullSlots' semaphore */
                INVOKE SemObj.Create(0, FullSlots)
                ON FAILURE {
                        RETURN FAILURE(Q_SEM_INIT_FAIL);
                };

                /* allocate and initialize the 'EmptySlots' semaphore */
                INVOKE SemObj.Create(Q_SIZE, EmptySlots)
                ON FAILURE {
                        RETURN FAILURE(Q_SEM_INIT_FAIL);
                };

                /* initialize the queue */
                qhead = qtail = 0;
                init = TRUE;
        };
```

Figure 8: Example Object Type Specification

```
OPERATION Insert(IN char: chr)
        /*
         * Take a character and insert it in the queue if it is not full.  If the queue is full, block the
         *  thread until space is available and then insert the character into the queue and return
         *  'SUCCESS'.  Return a failure indication if the queue is not initialized or if any of the
         *  invocations on semaphores fail.
         */
        {
                /* check if the queue has been initialized */
                if (init != TRUE) RETURN FAILURE(Q_UNINIT);

                /* get an empty slot */
                INVOKE EmptySlots.P()
                ON FAILURE RETURN FAILURE(Q_EPFAIL);

                /* insert a character into the queue */
                queue[qhead++] = chr;
                qhead %= Q_SIZE;

                /* produce a full slot */
                INVOKE FullSlots.V()
                ON FAILURE RETURN FAILURE(Q_FVFAIL);
        };

OPERATION Remove(OUT char: chr)
        /*
         * Remove a character from the queue, if it is not empty.  If the queue is empty, block the
         *  thread until a character is available and then remove one from the queue and return
         *  'SUCCESS'.  Return a failure indication if the queue is not initialized or if any of the
         *  invocations on semaphores fail.
         */
        {
                /* check if the queue has been initialized */
                if (init != TRUE) RETURN FAILURE(Q_UNINIT);

                /* get a full slot */
                INVOKE FullSlots.P()
                ON FAILURE RETURN FAILURE(Q_FPFAIL);

                /* remove a character from the queue */
                chr = queue[qtail++];
                qtail %= Q_SIZE;

                /* produce an empty slot */
                INVOKE EmptySlots.V()
                ON FAILURE RETURN FAILURE(Q_EVFAIL);
        };
}; /* end of the queue object */
```

Figure 8, continued

3.2.2 Locks

In addition to semaphores, which may enforce the mutual exclusion of threads to portions of code within objects, the kernel provides an abstraction with which to control thread access to specific regions of data within objects. This function is provided by the kernel-level concurrency control abstraction known as a *lock*. In Alpha, locking is the means by which objects control access to the data they encapsulate. By locking only those data that are being manipulated by a thread within an object, greater concurrency can be obtained than through enforcing the mutual exclusion of threads on regions of code. This is because different pieces of data can be manipulated concurrently by each thread executing a particular piece of code, allowing the possibility that multiple threads could execute the same sections of code concurrently without interference. The lock abstraction provides a means of controlling the concurrency of threads based on the data they need to access at any point in time.

An object needing to synchronize access to its data would allocate a ***Lock*** object, specifying as a parameter the data region with which to associate the lock. When the specified data is to be accessed, a thread must first acquire the lock (i.e., obtain the ***Lock*** object's permission to access its associated data). The desired operation is then performed on the locked data, and finally, the lock is released.

When a ***Lock*** object is created, it is associated with a contiguous span of an object's virtual address space. Similar to semaphores, locks are created and deleted by invoking an operation on the kernel-provided lock manager object. The data that a lock is to protect is defined when the ***Lock*** object is created, by passing the address of the start of the data to be protected, along with the data block's size. The lock manager object will respond with an error if the specified data region is invalid (e.g., if the data are not in the data portion of an object, are in non-existent memory, or overlap with another lock's region).

Once created, a lock is used to mediate all access to its specified data—all threads wishing to access the data must first be granted permission by the ***Lock*** object. Conceptually, the act of requesting a lock represents the application's indication to the system of its intention to access the data in a particular fashion (e.g., to write a new value to some data element), and the granting of a lock represents the system's notification to the application that the conditions for the desired manipulation of the data have been met (e.g., no other thread has write permission for the data).

A ***Lock*** object allows a thread to manipulate its associated data (i.e., grants the thread a lock) by allowing its virtual progress to continue following the completion of the thread's lock request operation. A lock suspends the execution of (i.e., blocks) a thread when its lock request conflicts with already granted requests for the same lock. For example, if one thread has already been granted an *exclusive access* lock† on a data item, other threads will block when attempting to lock the data, and cannot be unblocked until the thread currently holding the exclusive access lock releases it, thus allowing another thread to acquire a lock for the data region.

A ***Lock*** object has operations defined on it for: locking data items (LOCK), conditionally locking data items (C_LOCK), unlocking data items (UNLOCK), and modifying locks already held on data items (LOCK_CONVERT). A ***Lock*** object's LOCK operation is used to indicate a thread's desire to access the data associated with the lock. The C_LOCK operation is similar to the LOCK operation, except it does not block the thread if the lock cannot be granted; instead, it returns a

†*The modes in which a lock can be acquired will be described in the following paragraphs.*

status indication of whether the lock was granted or not. The UNLOCK operation is used by a thread to indicate that its manipulation of the locked data region is now complete. The LOCK_-CONVERT operation is used by threads to modify the mode in which a lock is being held, without first releasing it.

Locks in Alpha accommodate the fact that there are different types of data manipulations that can be performed. Rather than simply enforcing mutually exclusive access to data within objects, Alpha provides the potential for increased concurrency by using different types of locks to reflect the different types of manipulations that may be performed on the data to be locked. For example, it is frequently the case that multiple reads can be allowed simultaneously on a data item, without requiring synchronization among the threads performing the accesses.

To express the types of data manipulation that could be performed, and which of these manipulations are compatible, the Alpha kernel employs the notions of *lock modes* and *lock compatibility tables* [Bayer 79]. A lock mode specifies the kind of access that a thread intends to perform on the data associated with the lock. A lock compatibility table specifies which lock modes are compatible with other, currently granted, lock modes. A lock is termed *compatible* with another lock (i.e., the locks do not *conflict*) if the data accesses defined by the lock modes can be meaningfully performed concurrently. The lock modes defined in Alpha are:

- **Concurrent Read —** allows multiple readers of the data associated with the lock.
- **Concurrent Write** — allows multiple readers and multiple writers of the data associated with the lock; the lock holder may also read the data associated with the lock.
- **Exclusive Read** — only one thread can have a read lock on the data associated with the lock.
- **Exclusive Write** — only one thread can have a write lock on the data associated with the lock, and multiple readers are allowed; the lock holder may also read the data associated with the lock.
- **Exclusive Read/Write** — provides complete mutual exclusion to the data, only one thread can have access to the lock's data.

The compatibility table for locks in the Alpha kernel is shown in Table 1.

The Alpha lock facility has a number of features in common with the semaphores provided by the kernel. As is the case with semaphores, the language or operating system that serves as the kernel's client would ideally make the locking of data within objects implicit. However, it is only with the knowledge of the semantics of the data manipulation in question that locking can attain maximum concurrency. Upgrading (or promoting) lock modes can also lead to deadlock, and automatic lock request generation can exacerbate this problem. Most simple attempts at compile-time lock generation result in less than optimal concurrency. The Alpha locking mechanism is also accompanied by the possibility of deadlock, and as with semaphores, the lock mechanism can be used in conjunction with thread time constraints to aid in recovery from deadlocks (as well as livelocks and other failure conditions where computations are not making expected progress). Furthermore, as with ***Semaphore*** objects, capabilities for ***Lock*** objects cannot be passed to other objects, requiring that ***Lock*** objects can only be manipulated by the object that created them.

Requested \ Granted	none	Concurrent Read	Concurrent Write	Exclusive Read	Exclusive Write	Exclusive Read/Write
Concurrent Read	Compatible	Compatible	Compatible	Not Compatible	Compatible	Not Compatible
Concurrent Write	Compatible	Compatible	Compatible	Compatible	Not Compatible	Not Compatible
Exclusive Read	Compatible	Not Compatible	Compatible	Not Compatible	Compatible	Not Compatible
Exclusive Write	Compatible	Compatible	Not Compatible	Compatible	Not Compatible	Not Compatible
Exclusive Read/Write	Compatible	Not Compatible	Not Compatible	Not Compatible	Not Compatible	Not Compatible

— Compatible — Not Compatible

Table 1: Lock Compatibility Table

In addition to its role in concurrency control for objects, the lock abstraction plays a central role in the implementation of atomic transactions in the Alpha kernel. Some aspects functions of the lock abstraction in Alpha are specifically provided to support atomic transactions. The attribute of serializability typically associated with atomic transactions can be attained through the use of a two-phase locking discipline [Eswaren 76]. The Alpha kernel employs an optimistic strategy in supporting the atomic update of modifications made by atomic transactions—each lock has a write-ahead log associated with it to allow the changes made to the locked data item to be undone in case an atomic transaction aborts. Details of the manner in which locks are used in atomic transactions will be presented in the following chapter.

The current hardware does not adequately support the notion of locking in Alpha, and so the system software is responsible for performing much of the functionality associated with locks (e.g., detecting lock violations and enforcing the use of locks). This is an area where hardware support can provide great value to the system in terms of enhanced performance and improved functionality, and a research effort to construct the necessary (general-purpose) hardware support functions is currently underway.

An example of the use of locks is shown in Figure 9. This is similar to the queue example used previously, but now the ***Queue*** object's queue data and head and tail pointers are locked to ensure that concurrently executing threads do not corrupt the queue's logical structure or contents. By locking the queue pointers, consistency can be guaranteed with a greater degree of concurrency than is achievable through simple mutually exclusive access to the object's code. Note that in this case it is doubtful that the marginal increase in potential concurrency would offset the overhead associated with performing the locking operations. However, in more complicated situations the potential benefits of additional concurrency are much more significant.

```
/*
* An Example Queue Object Type Specification
*/
OBJECT Queue() {

/*
* Declarations
*/
#define Q_SIZE          100

static char             queue[Q_SIZE];
static int              qtail,
                        qhead;
WELLKNOWN CAPA          LockManager          LockObj;
CAPA                    Lock                 QueueLock,
                                             HeadLock,
                                             TailLock;

/*
* Operations
*/
OPERATION Initialize()
        /*
         * Initialize the queue object by allocating the locks, setting the queue pointers,
         *  and then returning successfully.
         * Return a failure if the allocation of locks is unsuccessful.
         */
        {
                /* allocate the queue data structure's lock */
                INVOKE LockObj.Create(QueueLock, queue, Q_SIZE);
                ON FAILURE {
                        RETURN FAILURE(Q_LOCK_INIT_FAIL);
                };

                /* allocate the head and tail pointer locks */
                INVOKE LockObj.Create(HeadLock, &qhead, sizeof(qhead));
                ON FAILURE RETURN FAILURE(H_LOCK_INIT_FAIL);

                INVOKE LockObj.Create(TailLock, &qtail, sizeof(qtail));
                ON FAILURE RETURN FAILURE(T_LOCK_INIT_FAIL);

                /* initialize the queue */
                qhead = qtail = 0;
        };
```

Figure 9: Example Lock Usage

```
OPERATION Insert(IN char: chr)
        /*
         * Take a character and insert it in the queue, if it is not full.  If the queue is full,
         *  return a failure indication.
         */
        {
                /* check if the queue is full */
                INVOKE HeadLock.Lock(CONCURRENT_READ);
                INVOKE TailLock.Lock(CONCURRENT_READ);
                if (((qhead + 1) % Q_SIZE) == qtail) {
                        INVOKE HeadLock.UnLock();
                        INVOKE TailLock.UnLock();
                        RETURN FAILURE(Q_FULL); };
                INVOKE TailLock.UnLock();

                /* bump the head pointer, and roll over if necessary */
                INVOKE HeadLock.Convert(EXCLUSIVE_READ_WRITE);
                ++qhead %= Q_SIZE;

                /* lock the queue, insert a character into it, and then unlock it */
                INVOKE QueueLock.Lock(CONCURRENT_WRITE);
                queue[qhead] = chr;
                INVOKE QueueLock.UnLock();
                INVOKE HeadLock.UnLock();
        };

OPERATION Remove(OUT char: chr)
        /*
         * Remove a character from the queue, if it is not empty.  If the queue is empty,
         *  return a failure indication.
         */
        {
                /* check if the queue is empty */
                INVOKE TailLock.Lock(EXCLUSIVE_READ_WRITE);
                INVOKE HeadLock.Lock(EXCLUSIVE_READ_WRITE);
                if (qhead == qtail) {
                        INVOKE TailLock.UnLock();
                        INVOKE HeadLock.UnLock();
                        RETURN FAILURE(Q_EMPTY); };
                INVOKE HeadLock.UnLock();

                /* lock the queue, remove a character from it, and then unlock it */
                INVOKE QueueLock.Lock(EXCLUSIVE_READ);
                chr = queue[qtail];
                INVOKE QueueLock.UnLock();

                /* bump the tail pointer, and roll over if necessary */
                ++qtail %= Q_SIZE;
                INVOKE TailLock.UnLock();
        }; /* end of the queue object */
```

Figure 9, continued

4 Key Concepts and Features

This chapter provides a description of some of the most significant features of the Alpha programming model which are not directly associated with a system abstraction, but rather have to do with the interactions among the kernel's basic abstractions.

Described here are the timeliness model and the approach used to meet the demands of time-critical applications, the different types of exceptions that can occur in a distributed, real-time system, the method employed to handle them, and the extensions to the kernel's basic abstractions that have been constructed in order to achieve a greater degree of system robustness.

4.1 Time Constraints

The fundamental programming abstractions of Alpha were specifically designed to support the system's overall objectives of global, dynamic, time-driven resolution of contention for system resources (e.g., processor cycles, communication bandwidth, memory space, or secondary storage). In particular, the thread abstraction provides a framework for injecting the application's time constraints into the system, and a basis upon which application-specific system resource management policies can be defined. Threads provide a unified means of managing all resources in the system—both within and among the processing nodes in the distributed system.

Many of the computations in a real-time application have time constraints associated with them, and because threads are Alpha's representation of these computations, threads also have time constraints associated with them. In order to provide globally consistent time-driven resource management, these application-specified time constraints for threads are carried along with the threads as they move through objects across the system's nodes. The timeliness constraints of threads are expressed in a form that provides a collection of application-specific timeliness information to the system and permits a wide range of time-driven resource management policies to be implemented. Time constraints are applied to threads in a nested, block-structured fashion. The policy currently used in managing resources takes full advantage of the information provided by the threads' time constraints in order to achieve a number of benefits over more conventional policies.

4.1.1 Time-Value Functions

Effective time-driven management of system resources in Alpha depends on the correspondence between the programmer's and system's view of application computations (provided by threads), and application-specified importance and time constraint information (provided by attributes of threads). Alpha uses a novel and effective technique for explicitly, and expressively, manifesting an application's time constraints: as the time-dependent value to the system of completing specific computations.

One thread attribute is the *importance* of the computation being performed by the thread, relative to the other threads in the system. This attribute is given to a thread when it is created and may be altered throughout the course of the thread's existence. The importance attribute of a thread is an application programmer's indication to the system of a computation's significance. Unlike a traditional priority, thread importance is not used in Alpha to indicate the manner in which resources (particularly processor cycles) will be apportioned to the thread. Instead, the timeliness information dictates the manner in which system resources are dispensed, and a thread's importance is only to be used in resolving contention for resources when insufficient resources exist to meet all of the demands for them.

The timeliness information associated with threads includes: the expected completion time for the execution of a block of code; the value (with respect to time) of completing the execution of the block of code; and a probability distribution function that indicates the likelihood of completing the block of code at any given time. Collectively, these attributes are the Alpha system's manifestation of *time-value functions*, which represent the time-varying value to the system of the execution of a computation (or parts thereof). Time-value functions allow the system to distinguish between a computation's timeliness and its importance, and represent a dynamic, powerful, and expressive notation for capturing time constraint information.

In the current implementation of Alpha, time-value functions consist of several different components, as illustrated by the example function shown in Figure 10. In this figure the horizontal axis represents real time, and the origin is the point at which a thread acquires the attributes represented by this time-value function. The vertical axis in Figure 10 represents the value to the overall system of completing the portion of a computation to which this time constraint applies. This value is expressed in globally meaningful value units.

To simplify the specification of time-value functions for practical applications, the full generality of the conceptual time-value function is not used. At this time, it is not considered necessary to allow arbitrary time-value functions to be specified at run-time. Instead, experience has shown that the selection of a critical instant in time, along with pre- and post-critical time function shapes from a parameterized collection of curves is more than adequate. By using a thread's importance attribute, its critical time and specified importance modifiers, the system can generate time-value functions that are capable of expressing a wide range of application timeliness constraints. As shown in Figure 10, each curve is scaled by the thread's importance, and the importance modifiers allow discontinuities to be introduced into the time-value function at the critical time. The particular curves provided by the system are defined at system-build time, along with the available resource management policies—e.g., the scheduling policy, as a part of the scheduling subsystem's policy specification.

When associated with threads, time-value functions provide the information necessary to execute a wide range of time-driven resource management policies in Alpha. The information provided by the time-value function not only provides sufficient information to meet the (rather substantial) needs of the *best-effort* processor scheduling algorithm (described in detail in the following subsection), but it also provides the information necessary to support round-robin, priority, rate-monotonic, shortest-processing-time-first, deadline, and slack-time processor scheduling algorithms, among others.

$\boldsymbol{I}$	—	thread importance
$\boldsymbol{t}_c$	—	critical time
$\boldsymbol{t}_e$	—	expected execution time
$\sigma(\boldsymbol{t}_e)$	—	standard deviation of $\boldsymbol{t}_e$
$\boldsymbol{S}_{pre}$	—	shape of value function, pre-critical time
$\boldsymbol{S}_{post}$	—	shape of value function, post-critical time
$\boldsymbol{i}_{pre}$	—	importance modifier, pre-critical time
$\boldsymbol{i}_{post}$	—	importance modifier, post-critical time

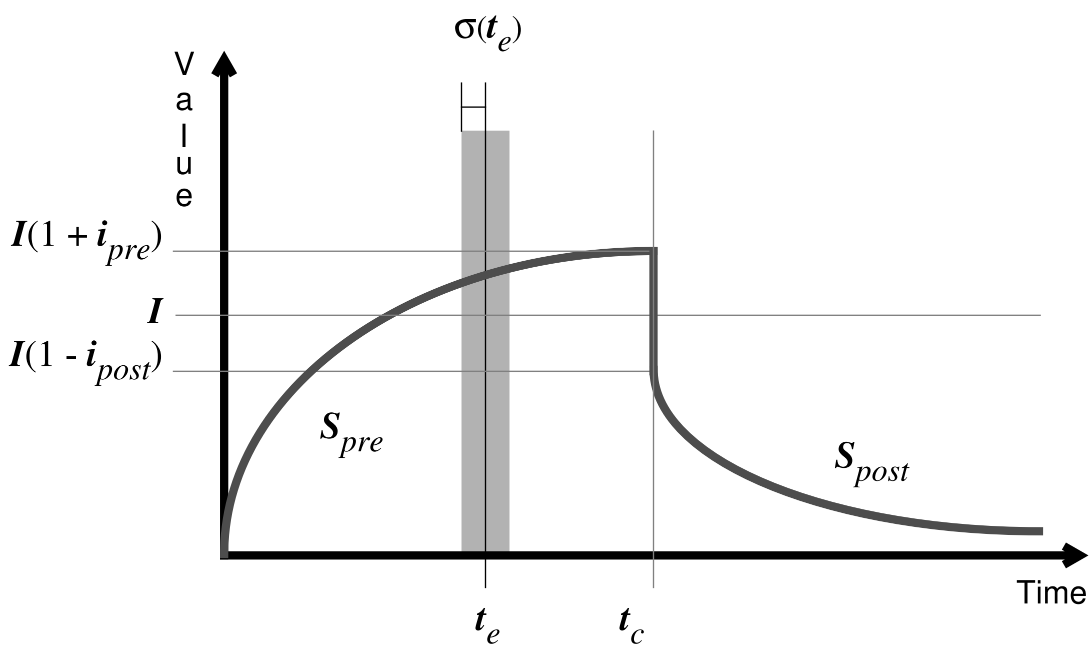

Figure 10: Components of a Time-Value Function

4.1.2 Time Constraint Blocks

Timeliness attributes are first associated with threads when they are created, and may be modified at run-time. In order to modify a thread's attributes, an invocation is performed on the thread itself, invoking the system-provided operation that allows the thread's attributes to be modified. When time constraint attributes are applied to a thread, the kernel can indicate, at run-time, the probabilities of successfully meeting a given time constraint. This feature provides an early indication of an exception and permits the application to determine the proper course of action for each individual case.

The notion of a *time constraint block* in Alpha provides a convenient way for the programmer to define a time constraint for the execution of a specific block of code within an object. Of course, the block may include invocations causing the thread to visit other objects. Nonetheless, the time constraint is in effect for the thread no matter where it goes until the block is completely executed (or the thread fails to complete the block in time). As with all thread attributes, Alpha permits the application of multiple, nested time constraints to threads. At all times, all of the time constraint blocks currently associated with a thread are in effect simultaneously (i.e., a nested timeliness attribute does not hide other timeliness attributes that are still in effect).

Figure 11 provides an example of the application of time constraint blocks to threads. The example is written in the C programming language, with extensions provided to simplify the act of programming the kernel interface directly.

There are a number of problems yet to be resolved concerning the application of time constraints to threads. These problems are the subject of on-going thesis research projects described in [Clark 88b] and [Maynard 88].

4.1.2.1 Dynamic Application of Time Constraints

Alpha allows the programmer flexibility in providing the parameters of time constraint blocks. For example, the time constraint block parameters may be defined at compile time (i.e., *early binding*), or they may be defined whenever a thread enters the defined block of code (i.e., *late binding*). Early binding is useful where it is desired that the code within a time constraint block must be executed by a certain, constant time, regardless of which thread is executing the instructions (e.g., when the combination of reading a sensor and moving an actuator must be performed within a certain amount of time). Late binding is useful where, although a block of code has a time constraint, the exact values of the time constraint parameter must be computed each time before entering the block (e.g., when the time required to complete a testing operation is dependent on the velocity of a specimen past a sensor). Furthermore, there are different types of late binding that can be performed. On the one hand, the parameters for a time constraint block may be a function of the global state of the object alone, in which case the time constraint would be the same for any thread executing the time constraint block. Alternatively, the time constraint block parameters may be a function of the local state of a thread (i.e., stack variables or incoming parameters) as well as the global object state, in which case the time constraint parameters for the block of code would potentially vary for each thread executing it. This flexibility allows the programmer to more dynamically and accurately represent the timeliness needs of a computation to the system, thereby making it possible for the system to manage resources more effectively.

4.1.2.2 Dealing with Unsatisfied Time Constraints

The very fact that time constraints exist and must be satisfied with finite resources implies that perhaps all of the time constraints cannot be satisfied. In that case, the programmer must deal with the effects of unsatisfied time constraints.

A time-value function's critical time indicates the nominal (i.e., desired) time of completion for the computation. The portion of the time-value function following the critical time may define a point at which the computation no longer has any value, and so in some sense such a time-value function represents a *deadline*—the time after which the completion of the computation no longer has positive value.

```
/* An Arbitrary Object Type Specification */
OBJECT Obj_A() {
        /* the object's first operation */
        OPERATION Opr_1(parms) {
                /* start of the operation's code */
                …
                /* enter this operation's first time constraint block */
                TIME_CONSTRAINT_BLOCK(parm1, parm2, parm3) {
                        /* start of code that is to be executed under this time constraint */
                        …
                        /* an arbitrary conditional statement */
                        if (condition) {
                                …
                                /* enter a nested time constraint block */
                                TIME_CONSTRAINT_BLOCK(parm1, parm2, parm3) {
                                        /* code to be executed under the nested time constraint */
                                        …
                                }
                                ON_ABORT {
                                        /* code to be executed if the time constraint is missed */
                                        …
                                } /* end of the nested time constraint block */
                                …
                        else {
                                …
                        }
                        …
                } /* end of the first time constraint block */
                …
                /* enter this operation's next time constraint block */
                TIME_CONSTRAINT_BLOCK(parm1, parm2, parm3) {
                        /* start of code that is to be executed under this time constraint */
                        …
                } /* end of the first time constraint block */
                …
        } /* end of the first operation */
        …
} /* end of the object */
```

Figure 11: Example Use of Time Constraint Blocks

Whenever a thread's time-value function is no longer positive (e.g., the thread has failed to meet its deadline), the thread's execution within the time constraint block should be terminated. This action is taken because the thread's time constraint has not been met, and its continued execution consumes resources and interferes with other threads operating under time constraints. Furthermore, the work that the thread continues to do will probably require more effort to compensate for or undo the manipulations of the object's data in order to restore it to a consistent state.

The fact that the continued execution of threads whose deadlines have been missed may have undesirable consequences demands that the system must put a stop to the execution of these threads in a prompt fashion. Simply halting a thread when its deadline has been missed is a totally unacceptable solution—in a real-time system, missing deadlines is as significant as defining and meeting deadlines. A much more meaningful thing to do when a deadline is missed is to redirect the execution of the thread to an exception handler, specified by the programmer, that dictates what must be done. The programmer's options might include: killing the thread, returning the object to a consistent state, reporting an exception to another object, compensating for the partial execution of the block, re-executing the block, logging an event, or just proceeding on.

Alpha supports the clean-up of computations which fail to satisfy their time constraints, to avoid wasting resources and executing improperly timed actions. This is done by immediately and forcibly diverting the normal flow of the thread's control to a defined exception handling block of code. By providing the programmer with a way of going to a defined point in his code on exception with all of the state information of the computation, a wide range of user-definable exception handling policies can be implemented. This is to say, a time constraint block has a single entry point, but has two exit points—one for normal exits (i.e., the necessary computation was completed before the specified time), and one for abnormal exits (i.e., the system indicates that the computation cannot be completed in the required time). A thread exits a time constraint block for one of three reasons: the thread has completed executing the code (and is exiting normally); the thread was aborted pursuant to a resource management decision by the scheduler (e.g., a missed time constraint); or the thread was aborted for some other reason (e.g., it was in an atomic transaction that has been aborted).

A time constraint is valid as long as the thread is executing code within the corresponding time constraint block. This implies that once a thread's execution is directed to a time constraint block's exception handling code, the block's time constraint is no longer in effect and the thread assumes attributes of its next outermost time constraint block. In this way, all execution of exception handling code is done under the proper computational attributes (i.e., the system uses the next level of time constraints so that the thread's attributes properly correspond to the thread's state). This means that the system continues to manage resources in the desired fashion, regardless of expired time constraints. The manner in which exceptions are handled by time constraint blocks is essentially the same way that block-structured, nested, mechanisms are used for handling other types of exceptions in the system (e.g., atomic transactions).

4.1.3 Time-Driven Resource Management Policies

A significant feature of the Alpha programming abstractions has to do with the combination of the fact that threads maintain a strong correspondence between the program's view of a logical computation and the system's manifestation of these computations. This feature makes it possible for the client programmer to associate application-specific attributes with computations.

Because threads are the physical manifestations of computations, each instance of the Alpha kernel can explicitly track and manage the computations running local to it. Threads provide a means of efficiently and effectively resolving contention for system resources. Furthermore, the kernel can assign global priorities (or relative-value functions) to the computations that are active in the system. Therefore, the thread-based approach provides a number of potential benefits over other, more common computational models. A discussion of these benefits is to be found in Chapter 5.

The Alpha operating system was designed to support a range of real-time scheduling policies. The resource management policy that has received the most attention by the project so far is known as the *best-effort* policy, and is used primarily for the management of processor cycles. The best-effort policy is the most ambitious and demanding of all scheduling policies implemented in Alpha so far and serves as a forcing function, requiring the greatest functionality of the mechanisms in Alpha.

Time-value functions form the basis for resolving all contention for system resources in Alpha—e.g., in managing processor cycles, communication resources, secondary storage access, and synchronization primitives (i.e., locks and semaphores). User-specified time-value functions and run-time statistics maintained by the system for each thread (e.g., accumulated execution time) serve as the fundamental inputs to the time-driven resource management policy modules in Alpha.

The time-driven resource management policies in Alpha all follow a common set of general guidelines. The time-value functions for all contending threads are evaluated *collectively,* and then the threads are scheduled so as to maximize the total value accrued by the system for the entire foreseeable future. The urgency of the computations requesting system resources is considered first—when there are sufficient resources to do so, the resource requests are serviced in an order that ensures that all of the computations' time constraints are met. If there are not enough resources available to satisfy the time constraints of all contending activities, a "best effort" is made to handle the overload condition gracefully (as defined by an application-specified policy). For example, an overload policy might indicate that the system should shed load on the basis of activity importance, or that it should retard all response time performance proportionally to activity importance. In the current implementation of Alpha, the former approach is taken—i.e., contending requests are selectively denied, on the basis of the urgency and importance of the computations responsible for the resource requests.

4.1.3.1 A Best-Effort Scheduling Policy

While many different types of scheduling algorithms may be (and have been) inserted into the framework provided by the scheduling subsystem, the algorithm that has received the greatest amount of use and attention to date is known as the *best effort* scheduling algorithms. This algorithm is a product of the Archons project's long-standing research efforts in real-time scheduling and represents the first practical application of our research work in this area. The best-effort policy takes application-specified information concerning the timeliness constraints of applications (i.e., the threads' attribute information) and attempts to meet all the application's time constraints, adapting to unexpected events. When the demands for processor cycles exceed the available supply (i.e., not all time constraints can be met), the best-effort policy discards requests (i.e., omits ready threads from the scheduling list) in such a fashion as to maximize the value to the system of the threads that are being scheduled for execution.

The best-effort policy includes one particular overload handling sub-policy. Examples of other overload sub-policies include: discarding the least important threads from the set of ready threads; discarding threads to maximize the total number of threads whose time constraints are met; not discarding any threads, but instead having all threads miss their deadlines by some average amount of time (i.e., distributing the overload evenly across all ready threads); or not discarding threads, but having the ready threads miss their deadlines by an amount inversely proportional to their value to the system.

Other, more typical, scheduling algorithms do not attempt to deal with overload conditions properly (in fact, some policies exhibit particularly bad performance in overload cases). In recognition of the system's robustness requirements, the best-effort policy was designed to handle overload conditions gracefully, attempting at all times to maximize the global value (as defined by the application) of the execution of computations to the system.

The best-effort algorithm in Alpha was designed from an aperiodic point of view, in that no special case treatment is given to events that occur cyclically. Typically, schedulers use the cyclic nature of some applications to obtain analytical leverage in making specious claims of "guarantees," and aperiodic events are force-fit into the periodic mold. Alpha makes use of a more general solution where time constraints are considered in a uniform manner, and no special significance is given to an event that may happen to recur in a more or less regular time pattern. Each iteration of a cyclic event is treated as an independent instance of a time constraint applied to a section of code. This allows periodic and aperiodic activities to be handled in an integrated, uniform manner and yields a highly adaptive scheduling subsystem. With the best-effort scheduler, the application does not necessarily fail when an assumption concerning the periodicity of events is violated, nor is a reevaluation of all of an application's timeliness constraints required when a change is made in one section of code.

It should be noted here that the best-effort scheduler implemented in Alpha can be made to behave like a range of different schedulers, depending on the amount and type of information given to it by the application programs. For example, with the thread's critical time parameter alone, the best-effort scheduler can behave like a *deadline* or a *rate monotonic* scheduler. When given the expected execution time parameter alone, the best-effort scheduler behaves like a *shortest-processing-time-first* scheduler. With nothing but the thread's importance parameter, the best effort scheduler degenerates to a *priority* scheduler. Finally, without any of the thread's scheduling attributes at all, the best-effort scheduler runs threads like a *round-robin* scheduler.

The best-effort scheduler allows Alpha to be employed in conventional static applications where rate-monotonic or rate-group scheduling are traditionally applied and where guaranteed behavior is required. However, Alpha's best-effort scheduler has significantly greater capabilities for more general real-time command and control applications than other typical real-time schedulers.

4.1.3.2 Implications of Time-Value Functions

With the full information provided by the applications programs with time-value functions, the best-effort scheduler can manage applications processor cycles much more effectively than other scheduling algorithms. Time-value functions effectively distinguish between the urgency and importance of a computation, whereas these attributes are most commonly encoded into a simple, small-integer priority code in existing systems. Also, the association of time-value functions with threads allows a direct expression of an application program's time constraints, with no translation or transformation required between the specification of an application computation's timeliness requirements and the computation's physical implementation. Time-value functions are a versatile representation that can uniformly (i.e., without special cases or exceptions to the model) express many different types of timeliness constraints, including: hard time constraints (i.e., deadlines), soft time constraints, hard or soft execution-time windows, and delayed execution. Because time-value functions can be specified dynamically and time constraint parameters may be run-time variables, a time constraint applied to particular section of code can, over time, exhibit a varying degree of "hardness" (i.e., the value of continued execution following the computation's critical time may change) and a dynamic maximum value (i.e., the global value to the system of completing the computation may change with respect to time).

Furthermore, in Alpha there is no premature binding of timeliness attributes that would restrict the ability of the system to carry out certain policies or its ability to adapt to events that might occur between the time that the binding is done and when the system makes resource management decisions. For example, the timeliness attributes are maintained with threads in an unencoded form and are not compressed into a form such as priorities. This means that there is no loss of information due to transformations of timeliness attributes by the system. Furthermore, because the timeliness attributes of threads can be dynamically modified throughout the course of a computation, Alpha can be more adaptive than systems which only support statically defined time constraints for computations.

4.2 Exception Handling

The overall system requirements of Alpha place certain essential requirements on its exception handling. Some of these exception handling requirements are common to all operating systems, some result from the underlying distributed system architecture, and yet others arise directly from the system facilities provided by Alpha. As a result, Alpha fields a wide range of different types of exceptions, all handled uniformly by a single, simple, exception handling facility.

The exception handling mechanisms in Alpha must meet certain constraints: they must fit well into the overall programming model (i.e., they should allow exceptional behavior to be handled without introducing new programming abstractions or severe aberrations in the existing abstractions), and their actions should be compatible with the system's time-driven management of system resources.

The operating system's client must be provided with mechanisms that allow the creation of applications that behave dependably, in the face of the entire range of exception conditions that may be encountered in the course of execution. In particular, the client must be able to specify the exact recovery actions to be taken should an exception occur while executing at a particular point in an object. To permit this, Alpha supports a mechanism that allows the client to associate application-specific recovery code blocks with individual regions of code within object operations, known as *exception blocks* (i.e., a form of "recovery blocks" [Anderson 81]). The client defines both the scope of an exception block and the exception handling code that is to be invoked should an exception occur when a thread is executing within the specified exception block. Without mechanisms that provide the client with the ability to define the location of exception blocks and application-specific exception handling code, the system is only able to perform crude, brute-force types of recovery activities (e.g., define exception blocks on operation or subroutine boundaries and in case of an exception, restore the object to a fixed, or otherwise known, state). Not only are Alpha's exception blocks more sophisticated, they are also more powerful since recovery may involve more than simply altering data values—in a real-time system, recovery may also involve changes that must be made to the external world. Of course, only the application programmer knows what these recovery actions should be, and it would be difficult for the programmers to inform the system about these actions if exception blocks were not provided.

In addition, the Alpha exception handling mechanisms must be asynchronous in their behavior. It is not acceptable in the Alpha real-time environment to wait until a thread makes a system call in order to notify it of an exception—in general, the longer a thread executes after an exception occurs, the more resources are wasted and the greater the effort needed to recover from the exception. For this reason, the exception mechanisms in Alpha must allow a thread to be notified of an exception immediately upon the detection of the exception at the thread's node, regardless of whether the thread is currently executing or if it is blocked at the time.

Furthermore, the system must have a means of ensuring that the consistency of system resources is maintained in spite of the exception behavior of threads. In the course of a thread's execution, it may acquire system resources (e.g., memory and synchronization tokens). Should an exception occur while a thread is in possession of system resources, the thread's normal flow of execution will be interrupted and both objects and the system may be permitted to remain in an inconsistent state. Regardless of the type of exception which occurs, the system should not be left in an inconsistent state, otherwise system resources could be lost (i.e., the system would be "leaky"). Thus, the system must keep track of the system resources acquired by a thread in order to be able to restore them to a consistent state in the event of an exception.

4.2.1 Mechanisms

In support of the robustness requirements of Alpha (as defined in [Northcutt 88a]), there are features of the operation invocation facility that are explicitly intended to support exception handling. In addition, the kernel provides an exception handling mechanism that provides the client with a block-structured programming construct for dealing with the various types of exceptions that might be encountered by an Alpha application. This exception handling mechanism augments the exception handling features provided by the operation invocation facility and deals uniformly with all forms of machine-, system-, and user-defined exceptions. The block-structured exception handling mechanism encompasses the time constraint block construct supported by the Alpha kernel, and provides a unified means of handling expired hard time constraints, aborted atomic transactions, orphaned thread sections, user-defined exceptions, and various machine exceptions (e.g., divide by zero, invalid memory reference, and access protection violations).

4.2.1.1 Operation Invocation

The Alpha kernel's operation invocation facility has, as a part of its basic definition, features that contribute to the system's reliability requirements and simplify the task of exception handling for the client. In particular, the semantics of operation invocation in Alpha guarantee that an invocation will behave much as a simple (local) procedure call, with a success or failure indication for each invocation.

An operation invocation can fail for a number of reasons—e.g., the node on which the target object exists fails, the communications link to the target object's node fails†, the target object specification is incorrect (i.e., due to an invalid capability, inability to locate the given object, etc.), or the invocation parameters are incorrect (i.e., wrong number of parameters, parameters are of the wrong type, capabilities to be passed are invalid, etc.). Should any of these exceptions occur in the course of an operation invocation, the kernel returns an exception indication, along with a *syndrome* indication that specifies which of the exceptions occurred. The exception syndrome provided to the programmer indicates all of the (potentially independent) exceptions that occurred during the invocation.

The operation invocation facility's exception indication allows the client to determine the appropriate course of action to pursue when an operation invocation fails; the system provides a basic mechanism, while the specific policy is determined by the client on a per-invocation basis. As a result of a failed invocation, an application could chose to: retry the same invocation a number of times, invoke an operation on an alternate object, report the invocation's failure to another object, perform a recovery/compensation action, return a client-level failure indication to its invoker, ignore the exception and continue execution, and so forth.

The invocation facility's exception syndrome conveys to the user information concerning the specific cause(s) of an exception. In general, it is poor practice for an operating system to provide information to its clients that cannot be meaningfully interpreted or used at the client-level. Despite the fact that the information returned on invocation failures concerning the failure of system nodes or communications links may not be useful to most application programs (in that there is little that application code can do about such system-level failures), this information can be used by system-level resource management objects to perform such tasks as dynamic reconfiguration, performance monitoring, or fault location, and is therefore returned in operation invocation failure notifications.

Figure 12 provides an example of the occurrence of a *thread break*, which creates an orphan thread section. Initially the thread is intact (Figure 12a), however, following the failure of $node_j$, a thread break occurs (Figure 12b). After the orphaned thread section has been deleted and eliminated, the new head of the trimmed thread can continue execution (Figure 12c).

---

†*Depending on the exact timing of communications link failures, the system may not be able to determine whether the invocation succeeded. It is considered to be an exception in Alpha when the system is not certain that an invocation has completed successfully, and this condition is indicated in the system's response to the operation invocation.*

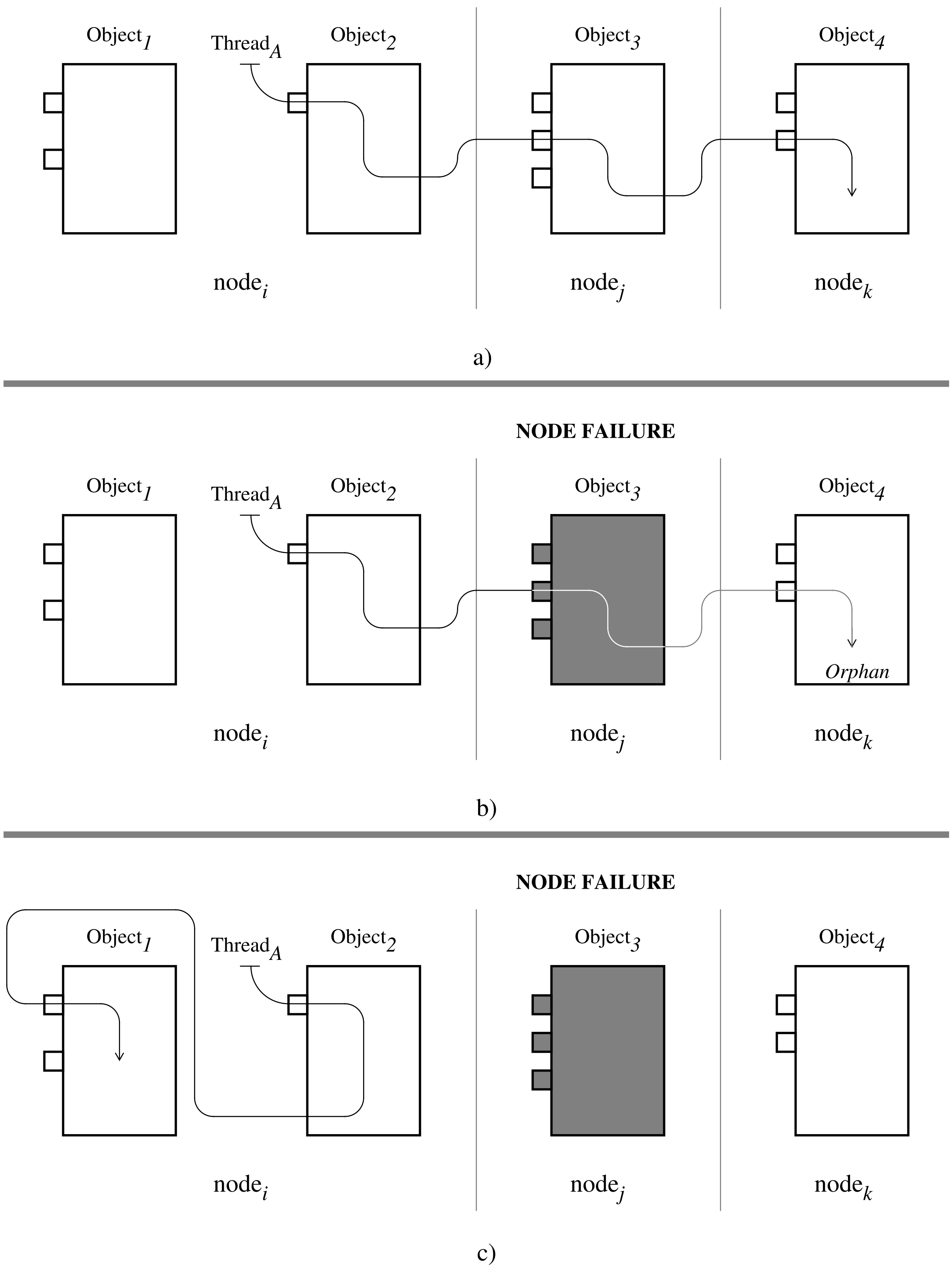

Figure 12: Example of Thread Maintenance

4.2.1.2 Exception Blocks

Alpha's kernel provides mechanisms which support the handling of machine-, system-, and user-defined exceptions through a programming structure known as an *exception block*. The exception block is a basic, uniform construct for handling such exceptions as aborted transactions, unsatisfied time constraints, and thread breaks. In addition the exception block facility is well integrated with the system's mechanisms for expressing and enforcing application-level time constraints.

An exception block is defined to be a region of code (as indicated by a begin/end pair of system calls) within an object operation, with an associated unit of exception handling code. The exception block is, in effect, a declaration on the part of the programmer that specifies the exception handling code to be executed should an exception occur while a thread is executing the block of code. This exception handling facility provides the programmer with a means of coping with the asynchronous exceptions that commonly occur within distributed real-time command and control systems. When such exceptions occur, the system forcibly diverts the execution of the affected thread to a well-known location (i.e., the user-defined exception handling code), with a predictable execution context (i.e., the variables, both global and local, within the lexical scope of the exception block). By allowing the client to provide application-specific exception handlers, it is possible for the system to perform specialized recovery/compensation actions on exceptions. This allows application objects to be restored to a consistent state in a much more efficient manner than can be accomplished by the brute-force techniques that must be used by the system in the absence of application-specific information.

The asynchronous diversion of a thread's flow of control can leave an object in an inconsistent state. The exception handling code that is associated with each exception block can be used to restore the consistency of the object's state, compensate for the effects of the aborted thread, or salvage usable results from the partially completed computation. When an exception occurs in Alpha, the objects affected are cleaned-up (i.e., made consistent) by executing each of the nested exception handling code blocks from the head of the thread, back toward the thread's root until the exception has been completely processed. This allows a thread to be cleaned up from within (i.e., by the thread itself, executing with its proper attributes). Furthermore, this activity can span multiple objects and nodes as the thread executes nested exception handling code blocks.

Exception blocks can be employed in a nested fashion. Therefore the occurrence of a single exception may require several levels of exception handling to be performed. For example, a thread may be executing within a series of nested exception blocks when the thread breaks, causing the orphaned head section(s) of the thread to be aborted. In this case, that curtails executing the exception handling code for the exception blocks that exist (on the nodes that are still running) between the current head of the thread, and the point in the body of the thread where the exception occurred (i.e., at the ON_FAILURE clause of invocation immediately preceding the thread break point). Conceptually, all of the orphaned thread fragments are cleaned up concurrently following an exception—i.e., all of the exception handling code blocks are executed concurrently with respect to each other. Furthermore, the system guarantees that all orphaned thread fragments are eliminated.

Whenever possible, the orphan's exception code blocks are executed, before the new head of the thread proceeds with its execution. During exception handling, the system adjusts the attributes of the thread sections in such a manner as to ensure that each exception code block is executed with attributes appropriate for the thread section at that point. Among other things, this ensures that the proper scheduling parameters are used while cleaning-up a thread following an exception. Furthermore, if an exception occurs while a thread is blocked (e.g., as a result of a thread break or atomic transaction abort), the thread head is vectored to its proper exception handling code block and made ready to execute (an error indication is returned if the thread is blocked waiting on an operation invocation). In addition, the kernel maintains a record of the thread's local state when an exception block is entered, and uses that record to restore the state when an exception occurs. In this way, the thread's execution state while within the exception handling code can be made to include all of the local and global variables that are at, or outside, the scoping level of the exception block.

In addition to any application-specific exception handling code that is specified with an exception block, the system performs its own, system-defined, exception handling. The system's exception handling code performs only the simple clean-up operations necessary to ensure that the system's resources are maintained in a consistent state. This ensures that, even if the application programmer does not provide any exception handling code, the system will remain consistent (i.e., it will not "leak" resources, nor will the system fail due to inconsistent internal data structures). To have each object behave in a fashion that guarantees the minimally acceptable degree of system consistency in the face of the various types of failures that may occur, an exception block is automatically placed around each operation within all objects in the system. Thus, regardless of the code contained within an operation, the exception handling facility can provide sufficient internal system consistency.

The robustness of Alpha is enhanced by optimizing the system design and implementation for the exceptional cases, instead of the expected ones. Examples of the application of this principle can be found in the system's exception handling mechanisms, communications protocols, and operation invocation facility. The Alpha exception handling mechanism provides a unified mechanism for the management of exceptions associated with time constraints, atomic transactions and machine exceptions. For this mechanism, the normal condition is that the transaction commits, the time constraint is satisfied, or no machine exception occurs; the exception case is that the transaction aborts, the time constraint is not satisfied, or a machine exception occurs. Performance is optimized for the exception case by trapping into the kernel on every exception block entry, in order to deposit the state information needed in case an exception occurs while executing within the block.

Another example is found in the remote operation invocation protocol, which is used to monitor and repair threads as they extend across nodes. The normal case for this protocol is that the thread is intact; the exception case is when a node or the communications network fails and the thread is broken. This mechanism's performance is optimized for the exception case (at the expense of the normal case) by the periodic exchange of keep-alive messages among the nodes that a thread spans, rather than using only end-to-end time-outs on the remote invocations.

Finally, the kernel does not make use of hints in any form. In particular, the internal global identifiers used to access programming entities (e.g., threads and objects) in Alpha do not include an explicit reference to the physical location of the entity. If hints were used, the expected case for this function would be that the hint is correct, while the exception case would be that the hint is wrong. In Alpha, however, the exception case's performance is optimized by performing a multicast-like message transmission on each remote invoke, which will be received by the addressed entity regardless of its physical location (which may change over time). This is done instead of using a hint indicating the entity's likely physical location to perform a point-to-point remote procedure call, which also includes a special sub-protocol that is performed to locate the target entity when the hint is wrong. While hints might speed references in a static system, when dynamic reconfiguration of the system is underway, the use of hints incurs a higher than normal cost when the system can least afford it—i.e., in an exception condition.

4.2.2 Example Usage

The exception block facility in Alpha is analogous to the UNIX signal handling facility. This exception handling facility represents the operating system's contribution to meeting basic reliability requirements, and is not intended to preempt or subsume language-specific exception handling constructs. The following provides an example of the use of the operating system's native exception handling facilities, and does not reflect any attempt to integrate these mechanisms with a particular programming language.

The exception block construct provides the programmer with a wide range of exception handling policy options. On exception, the exception handling code can attempt to restore an application object to a consistent state, it can perform compensating actions in order to account for the effects of partially completed operations, or it could make effective use of the (partial) results generated up to the point in time when the exception occurred.

Figure 13 provides an illustration of nested exception blocks that perform compensating actions to recover from exceptions in a more efficient manner than is possible using more brute-force techniques (such as the gross restoration of the object's complete global state). The object in the example maintains a collection of data records in two different lists, one unsorted and the other sorted. This example deals with an operation on the object that moves a given data record from the unsorted list to the sorted one.

The body of the example operation is encapsulated within an exception block in order to ensure that the consistency of the object is maintained in the face of thread failures. For this object, consistency is defined as the state where both of the list data structures are consistent and each data record in the object exists in exactly one list. The exception handling code ensures that, should an exception occur where the object remains intact (e.g., a thread break, transaction abort, or missed time constraint), the object will be restored to a consistent state. This compensating action takes advantage of the programmer's knowledge of the semantics of the operation being performed to restore the object's consistency at a much lower cost than that which would be imposed by the system (e.g., saving the state of the object in each consistent state, and then restoring the state on exceptions).

```
/* Object Type Specification */
OBJECT ObjectName() {
        /* the object's global data declarations */
        linkliststr        list1, list2;
                ...

        /* the object's local procedures */
        listelemptr        Pred(list, elementptr) {
                ...
        listelemptr        InsertPoint(list, elementptr) {
                ...
        listelemptr        EndPTr(list) {
                ...

        /* one of the object's operations */
        OPERATION Change(IN listelem elem)
        /*
         * This operation moves the element given as a parameter from list1 (where
         *  it is assumed to be) to list2.
         */
        {
                listelemptr        eptr;

                /* check if the given element is in list1 */
                if (InList(list1, elem) != TRUE) {
                        /* return an error signal */
                        RETURN FAILURE(NOT_IN_LIST1);
                }

                EXCEPTION_BLOCK {
                        /* remove the element from list1 and put it in list2 */
                        eptr = Pred(list1, &elem);
                        eptr->next = elem.next;
                        eptr = InsertPoint(list2, &elem);
                        elem.next = eptr;
                        eptr = Pred(list2, eptr);
                        eptr->next = &elem;
                }
                ON_EXCEPTION {
                        /* check if the element being moved is in one of the lists */
                        if !((InList(list1, elemptr) == TRUE) || (InList(list2, elemptr) == TRUE)) {
                                /* it is not, so put it at the end of list1 */
                                elempt->next = NIL;
                                eptr = EndPtr(list1);
                                eptr->next = elemptr;
                        }
                }
        } /* end of the 'Change' operation */

        /* more of the object's operations */
                ...
} /* end of the object */
```

Figure 13: Exception Handling Example—Compensation

Figure 14 provides an illustration of how the Alpha exception handling mechanism can be used in conjunction with the timeliness mechanisms to create operations that are limited in the maximum amount of time they can take to complete. In this example, a database search is performed to obtain a record that has a key that closely matches one given as an invocation parameter. The operation returns either a record with an exact key match, or the closest match found in the given amount of elapsed real-time. Note that in this example, a programmer is able to trade-off accuracy for time in the process of performing a complex, potentially time consuming operation. This strategy may be useful in such time-limited operations as: matching incoming plots to tracks within a track database; determining the next move to make in a chess game; and finding the closest match in a collection of patterns.

In Figure 15, the exception handling mechanism makes use of the partial results obtained prior to an exception caused by an expired time constraint. This example shows an operation that computes a number based on the combination of a collection of coefficients. This operation is limited in the amount of time it can take to complete, and if the complete set of coefficients is not made available in the specified amount of time, defaults are substituted for the missing values and the result is generated with a degree of accuracy that depends on the number of coefficients that were computed before time ran out. Note that in this example results are returned with an indication of the degree of confidence associated with the result values. This is an example of the type of operation that might be used in an application that performs computations on "fuzzy" (i.e., inaccurate or incomplete) data in order to achieve the best results possible within a given time limit.

```
/* Object Type Specification */
OBJECT ObjectName() {
        /* the object's global data declarations */
        elemstr         database[DB_SIZE];
                ...

        /* the object's operations */
                ...
        OPERATION TimedMatch(IN keystr key, IN float time, OUT elemstr elem)
        /*
         * This operation returns the record with the closest match to the given key that
         *  can be found in the given amount of time.
         */
        {
                int     depth;
                float   diff, mindiff;
                elemstr eptr, besteptr;

                /* initialize the local variables */
                depth = 0;
                diff = 0;
                mindiff = MAX_FVAL;
                besteptr = &nullelem;

                /* set the time limit for the search */
                TIME_CONSTAINT_BLOCK(time, EXP_RT(time), STD_DEV(time), STEP) {
                        /* try to find a good match in the database */
                        do {
                                eptr = Search(key, depth++);
                                diff = KEY_DIFF(eptr->key, key);
                                if (diff < mindiff)  besteptr = eptr;
                        } while (diff >KEY_THRESH);
                }
                ON_EXCEPTION {
                        /* use the best match found so far */
                        eptr = bestptr;
                }

                /* return the best match obtained by this search */
                elem = *eptr;
        } /* end of the 'TimedMatch' operation */

        /* more of the object's operations */
                ...
} /* end of the object */
```

Figure 14: Exception Handling Example—Time-Limited Operations

```
/* Object Type Specification */
OBJECT ObjectName() {
	/* the object's global data declarations */
	float	defcoeff[] = { … };
		…

	/* the object's operations */
		…
	OPERATION BigCompute(IN seedstr seed, OUT float answer, OUT float acc)
	/*
	 * This operation computes a value based on the combination of a set of coefficients
	 *  that are computed each time the operation is invoked.  The amount of time that this
	 *  operation can take is fixed, and default values are used for those not computed.
	 */
	{
		int	index;
		float	coeff[MAX_SIZE];

		/* set the time limit for the computation */
		TIME_CONSTAINT_BLOCK(TIME_LIMIT, EXP_RT, STD_DEV, STEP) {
			/* compute the necessary coefficients */
			for (index = 0; index < MAX_SIZE; index++) {
				coeff[index] = GenerateCoefficient(index, coeff, seed);
			}

			/* indicate the degree of accuracy acheived */
			acc = 1.0;
		}
		ON_EXCEPTION {
			/* compute the degree of accuracy */
			acc = (index / MAX_SIZE);

			/* use the default values for the rest of the coefficients */
			for ( ; index < MAX_SIZE; index++) {
				coeff[index] = defcoeff[index];
			}
		}

		/* compute the result using the coefficients */
		answer = GenerateResult(coeff);
	} /* end of the 'BigCompute' operation */

	/* more of the object's operations */
		…
} /* end of the object */
```

Figure 15: Exception Handling Example—Using Partial Results

4.3 Robustness

For the purposes of this work, a commonly used distributed system failure model was adopted [Anderson 81]: both hardware and software component failures are considered here, in both the system and application domains, including both *transient* and *hard and clean* failures.

The robustness techniques employed in Alpha are supported primarily by kernel mechanisms that provide a client interface at which failures in the underlying system are abstracted into a set of well-defined, predictable behaviors. In particular, the following robustness issues are addressed:

- **consistent behavior of actions**—provided by mechanisms that independently support the attributes associated with atomic transactions (i.e., atomicity, permanence, and serializability). These attributes are provided in the form of individual mechanisms in order to provide a range of levels of service at a range of costs, allowing applications to pay only for the amount and type of reliability needed.
- **availability of services**—provided by mechanisms that allow objects to be replicated and manage the different types of interactions defined on those replicas.
- **graceful degradation**—provided by mechanisms that use an ordering function (currently based on the timeliness constraints and relative importance) associated with all requests for services, in order to sacrifice lower-valued requests in favor of higher-valued ones when resource allocation conflicts arise.
- **fault containment**—provided by mechanisms that place each object in a separate (hardware-enforced) address space, and by separating software components into private system-enforced protection domains, with all interactions restricted to those explicitly allowed by the capability mechanism. This supports a form of defensive protection, where errors are prevented from propagating among objects.

While the Alpha kernel provides a set of mechanisms to support these objectives, its robustness mechanisms are not intended to form a complete facility. The kernel is intended as a framework within which policy issues relating to these robustness techniques can be explored. The mechanisms provided in Alpha for atomic transactions and replication are initial versions of the more complete mechanisms being developed in on-going research by the Archons project to develop system-level policies which use these mechanisms.

The Alpha operating system's concern for reliability is manifest at all levels within the system—from the basic assumptions, to the programming abstractions, and all the way down through the system's design and implementation. The variety of object-orientation in Alpha was chosen in the belief that it would be well-suited to the type of robustness techniques that have been developed by the Archons project for real-time command and control applications [Clark 88b, Sha 85]. The Alpha object model provides a disciplined control structure for interactions among software components (as compared to the less structured process and message-based system model) and restricts access to encapsulated data items. Among other things, this serves to simplify the task of tracking the operations performed on objects that is required in the implementation of atomic transactions. The fact that the object model centralizes all accesses to encapsulated data reduces the complexity involved in structuring operations so as to maximize the concurrency that can be obtained from objects (both within and outside of atomic transactions).

Operation invocation is controlled by the kernel through the use of capabilities. In this way, the ability of objects to invoke operations on other objects can be restricted to only that set of destination objects explicitly permitted. Capabilities can be given to objects when they are created, or they can be passed as parameters of operation invocations. In the kernel, the capability mechanism provides basic, defensive protection at a low cost in terms of performance.

Also in support of fault-containment, Alpha places objects and threads in separate, hardware protected address spaces, and the Alpha communications subsystem provides for the time-driven detection and elimination of orphaned threads (due to communication path or node failures).

Robustness in Alpha is enhanced by mechanisms in support of atomic transactions and object replication. Atomic transactions provide for the correctness of actions and the consistency of data in the face of node or link failures. Replication of objects provides the physical redundancy necessary to support the availability and performance requirements of Alpha.

4.3.1 Atomic Transactions

Atomic transactions have been shown to be very useful in the construction of reliable applications such as database systems [Eswaren 76, Lampson 81]. Some early reports proposed the use of transactions within distributed operating systems [Lampson 81, Jensen 84], and in recent times the belief that atomic transactions may prove useful within distributed operating systems has become more widely accepted. A number of efforts are currently underway to explore the inclusion of atomic transactions as an operating system service or as a language primitive [Popek 81, Liskov 84, McKendry 84, Almes 85].

Because the primary goal of the Alpha kernel is to support research into the development of distributed operating systems for real-time supervisory control, certain robustness constraints are implied—i.e., the kernel itself must function reliably in the face of system component failures, and the kernel must provide the application with mechanisms that will allow the application to function with similar robustness. While other research efforts have explored the notion of atomic transactions, few have attempted to include atomic transaction support within an operating system kernel. Typically these efforts provide atomic transaction facilities on top of existing operating systems. Atomic transaction support was one of the major factors that influenced the design of the Alpha kernel—in fact, it is one of the primary reasons that the object model was chosen as the fundamental programming paradigm.

By integrating the notion of atomic transactions into the design of the kernel, it is anticipated that the resulting performance of atomic transactions at the application-programming level should be sufficiently high to allow meaningful experimentation to be performed. Previous attempts at constructing atomic transactions on top of existing operating systems indicate that such an approach can degrade system performance to the point of making it difficult to implement meaningful applications. Furthermore, the inclusion of atomic transaction mechanisms within the kernel provides robustness support, similar to that provided to the application, for use within the system itself. The same benefits that atomic transactions bring to the construction of application programs is useful in the construction of system software.

The mechanisms currently provided by the Alpha kernel in support of atomic transactions are meant to be representative of the more comprehensive set of mechanisms being developed as a part of a doctoral research project currently in progress[Clark 88b].

4.3.1.1 Concepts

The notion of an atomic transaction, as commonly defined, encompasses a number of different concepts that provide a means of achieving a type of system behavior that is useful in constructing reliable systems. In the context of Alpha, classical atomic transactions may be viewed as a discipline applied to the use of mechanisms and a set of programming conventions that together result in making objects appear to behave in a well-defined manner despite the failure of system components. By enforcing this desired behavior on objects, it has been shown that reliable distributed applications may be constructed [Popek 81, McKendry 84, Almes 85]. In the Alpha kernel, the commonly accepted definitions of what constitutes well-behaved objects and what types of system failures will be considered are adopted as requirements for this work [Moss 85]. The intent of the Alpha kernel is to support these definitions as baseline requirements and also to provide a vehicle for the refinement of these definitions.

An atomic transaction is traditionally defined as a computation (possibly a part of a broader computation) that performs actions on objects, the effects of which appear to be done atomically with respect to failures and other transactions, and all transactions appear to execute independently of all others [Lampson 81]. In a simplified view, actions are typically characterized as reads and writes on data that are represented by objects. The major concepts that constitute the classical notion of atomic transactions may be summarized as follows:

- **Atomicity**: This is the property of the *all-or-nothing* behavior of transactions—i.e., either all of the individual actions comprising a transaction are successfully performed, or none of them is performed. The effect of atomicity is that (from an external view) the state of the system transitions from one consistent state to another. In this case, consistency is defined as some predicate on the data items, known as an *invariant*. While the database itself may be in an inconsistent state at some point in time, the property of atomicity ensures that this state is not externally visible. Atomicity therefore provides the guarantee that, despite failures of system components, no data object can be observed in a state that does not satisfy the system's invariant conditions.
- **Permanence**: This is a property of objects that ensures the continued existence of the (externally observable) effects of successfully completed transactions, even in the face of system component failures. Once a transaction reaches a successful completion and the effects of its actions are made visible to others, failures in system components will not result in the state of data objects reverting to some previous state.
- **Serializability**: This property of transactions deals with the relative ordering of actions among separate transactions. The individual actions comprising concurrently executing atomic transactions are executed by the system's processors in some partial order known as a *schedule*. A *serial schedule* is one where all of the actions of a particular transaction are executed either before or after all of the actions of any other transaction. A schedule is defined to be *serializable* if its effects are the same as if a serial schedule had been executed. Thus, serializability is the property of atomic transactions that provides the appearance of a non-interleaved execution of individual transactions.

4.3.1.2 Approach

Because of the real-time nature of the intended application domain for Alpha, the availability of system resources and services is of equal, if not greater, importance to providing for consistent restart after an arbitrary period of unavailability. In a real-time command and control system the quality of information tends to degrade over time. Of course, data stored on a failed node is typically unavailable while the node is down, and in fact when the node recovers, the stored information may be invalid, even if it is consistent according to some invariant. Thus, the definition of consistency in a real-time system must include a specification of time in addition to the normal system invariants. In these respects, Alpha differs from many other database-oriented applications that use atomic transactions (e.g., banking systems or airline reservation systems). This is why an atomic transaction facility plays a necessary, but not sufficient, part in meeting the system's robustness goals—because atomic transactions provide some, but not all, of the properties of good behavior that are needed in constructing reliable applications.

The atomic transaction mechanisms provided by the Alpha kernel are meant to be general mechanisms for use by the authors of both system and application code. The kernel does not enforce any policy on the use of these mechanisms, nor does the kernel automatically apply atomic transactions to client-defined code—the client chooses when and where atomic transactions are to be used.

The atomic transactions supported by mechanisms in the Alpha kernel may be *nested*. This allows atomic transactions to be placed completely within other atomic transactions. This is done to provide a finer level of granularity than can be achieved by placing all actions within one large, top-level atomic transaction. Nested transactions also provide a form of modularity in which an object may use atomic transactions to achieve its given level of robustness, and this use of atomic transactions is not visible to invoking objects. In a nested atomic transaction, if a lower-level atomic transaction fails it is reported back to the level that initiated the transaction, where it is decided by the client's code whether the transaction should be retried or whether this level should abort to the next level up.

In addition to providing the basic functionality of atomic transactions as described in the previous subsection, the Alpha kernel provides mechanisms to allow further research in the area of modular, high-concurrency transactions (based on related research [Allchin 83, Sha 85]). One particular area of atomic transaction research that the Alpha kernel is meant to support is the exploration of the notion of *compound transactions* [Sha 85]. A compound transaction is a form of atomic transaction that is designed to provide higher concurrency than a normal atomic transaction and minimize the problem of cascading aborts. This implies that the atomic transaction mechanisms provided by the Alpha kernel should permit the relaxation of serializability constraints and should permit the use of compensating actions as opposed to returning to previous versions in reaction to aborts. To reduce life-cycle costs, the atomic transactions supported by Alpha should also exhibit a high degree of modularity (i.e., clients should not be greatly inconvenienced when new operations or objects are to be added). Furthermore, since Alpha's application domain is real-time process control systems, the atomic transaction mechanisms provided by the kernel should bound the amount of time required for transactions to terminate (i.e., either commit or abort).

4.3.1.3 Mechanisms

The common definition of atomic transactions represents a single data point in a multidimensional decision space. The kernel of Alpha provides mechanisms that support a range of definitions of atomic transactions by providing some degree of movement along all of the dimensions of this decision space. The atomic transaction mechanisms provided by Alpha are not completely orthogonal and some points in this space are not meaningful, however these mechanisms represent policy decisions that do not restrict the exploration of the atomic transaction design space. The kernel does not force the client to use a particular form of transaction, but rather allows the client to choose when, where, and what type of atomic transaction to use, based on the functional requirements and their associated cost.

In the context of Alpha, atomic transactions can be thought of as forming (potentially nested) brackets around portions of threads. Each thread in the system may be executing within a nested atomic transaction or outside of any atomic transaction at any point in time. The kernel provides mechanisms that may be used by threads to define when a thread is to enter an atomic transaction and when the thread is to exit the atomic transaction (either by committing or aborting it).

In Alpha, the definition of atomic transactions is decomposed into three separate attributes each of which is supported by one or more mechanisms. These attributes are:

- **Permanence** — which dictates whether the secondary storage image of an object is maintained in volatile storage, where it does not persist across failures of the node that contains the object's primary memory image, or whether the secondary storage image is kept in non-volatile secondary storage, from whence it can be regenerated following the failure of its node.
- **Failure Atomicity** — which ensures that changes to the secondary storage image of objects are made atomically with respect to system failures. It requires that object updates be done in such a way as to allow the objects to be reconstituted in a consistent state after node failures. This attribute provides the *all-or-nothing* property of atomic transactions by governing the way in which the secondary storage image of an object is modified by atomic updates. This is the functionality typically implemented using stable storage mechanisms [Lampson 81].
- **Serializability** — which provides the appearance (to external observers) that all transactions execute in a non-overlapping serial order. It is not necessary that the actions of threads be actually serialized, but their effects need to be equivalent. This attribute involves the control of the visibility, both inside and outside of atomic transactions, of changes made to objects by threads. The attribute of serializability ensures that changes made to an object by a thread are not made visible to other threads until the transaction commits (in which case the changes are made visible to all threads) or aborts (in which case the changes are undone before being made visible).

The mechanisms in the Alpha kernel that support atomic transactions fall into three categories. Some mechanisms are provided solely for the support of atomic transactions, others are variations of (or extensions to) existing mechanisms, and yet others are general purpose mechanisms that are useful in implementing atomic transactions. In explicit support of atomic transactions is a kernel-provided transaction management object that has defined on it operations to begin, commit, and abort atomic transactions. A thread invokes the BEGIN_TRANSACTION operation on the transaction management object when it wishes to initiate an atomic transaction, either a top-level transaction or a nested transaction. Threads invoke the END_TRANSACTION operation on the transaction management object when they wish to commit the current level of atomic transaction. The ABORT_TRANSACTION operation is invoked on the transaction management object when a thread wishes to abort the atomic transaction that it is currently executing. A restriction (that can be enforced at compile-time) on the use of atomic transactions in

Alpha is that the operation invocations to begin, end, or abort a specific transaction must exist within the same operation in an object.

The general purpose mechanisms that are also used to support atomic transactions include the different object attributes (i.e., transient/permanent, atomic/non-atomic update), the thread concurrency control mechanisms, and the invocation mechanism. The permanent and atomically-updated object attributes are used to provide transaction permanence and failure atomicity, and the concurrency control mechanisms are used to provide transaction serializability. The invocation mechanism is used to perform the COMMIT operation on all of the instances of the transaction manager involved with a particular atomic transaction.

Each object in Alpha has defined on it a set of standard operations for pre-committing, committing, and aborting atomic transactions. A client may provide specialized COMMIT and ABORT operations or, if these operations are not specified by the client, the kernel-provided set of default operations is used. By providing custom COMMIT and ABORT operations, the client can define special functions (such as compensating actions) that make use of less expensive mechanisms or provide more appropriate behavior that reflect the semantics of the operations being performed in order to maximize performance. In cases where compensation can be done, it is possible to make use of this feature of Alpha to implement compound (or other non-serializable) transactions [Sha 85]. The default operation for transaction pre-commit prepares to write the committed state of the object to secondary storage by invoking a PREPARE_UPDATE operation on the object. The default COMMIT operation releases all of the locks and semaphores associated with the transaction being committed, and invokes a COMPLETE_UPDATE operation on the object. The default ABORT operation restores the pre-transaction state of all of the locked data items from their logs, releases all of the locks and semaphores associated with the transaction, and invokes a CANCEL_UPDATE operation on the object.

4.3.1.4 Usage

Typically, atomic transactions have the attribute of *permanence*, in the sense that failures can occur and the changes made to objects by committed atomic transactions remain in effect across failures of system components. This represents an extreme case that provides a high degree of robustness at a commensurately high cost in terms of performance. In Alpha, atomic transactions may have differing degrees of permanence associated with them, thereby providing less than total permanence at less than the worst-case cost. This means that a transaction may commit, but some types of failures will result in the effects of committed atomic transactions being lost.

The attribute of failure atomicity is supported by the object update mechanism and the invocation mechanism. By defining an object to be atomically updateable, the object takes on the attribute of changing states atomically with respect to failures. This function ensures that the secondary storage image of objects is always consistent, and is a necessary part of the failure atomicity attribute. In addition to providing atomically updateable objects, the kernel must be able to ensure that all or none of the actions contained within an atomic transaction commit. This atomicity function is supported by the Alpha's invocation mechanism. When a thread executing in a transaction breaks, in addition to the normal *thread repair* that is done as a part of all invocations, a function known as *visit notification* is also performed by the invocation mechanism on behalf of the thread that has been broken. Visit notification requires that the kernel track all of the objects visited by the thread while in a transaction, and signal these objects (along with the transaction management object) when the transaction commits or aborts.

Synchronization atomicity is attained by Alpha's concurrency control mechanisms. In the absence of some form of concurrency control there are no constraints on the visibility of changes to objects, and hence all changes made to an object are instantaneously visible to any thread executing in that object. The concurrency control mechanisms provided by Alpha allow threads to control the visibility of changes made to objects. For example, by inhibiting thread access to particular data, a thread may restrict visibility of changes it makes to the data until an atomic transaction commits. In many cases, by taking the semantics of an object's operations into account, a high degree of concurrency can be obtained by threads within objects. Furthermore, the client is free to use the visibility controls in such a way as to maximize use of the concurrency available in the implementation of objects.

The attribute of serializability is provided by enforcing a discipline on concurrent actions, through the careful use of concurrency control mechanisms. The locking mechanism is the primary means by which the serializability attribute is provided in Alpha. By applying a two-phase discipline to the use of locks, serializability of atomic transactions may be achieved. Two-phase locking is only one means by which the goal of serializability may be achieved. Once again, by considering the operations performed by an object, it is possible to achieve the desired effects of serializability at a lower cost in terms of performance.

4.3.1.5 Issues

Operations may be invoked on objects concurrently by threads (some of which may be executing transactions, while others may not). This results in the possibility that threads executing transactions will operate on data that has not been committed (i.e., transferred to the object's secondary storage image). In order to provide the attribute of serializability, despite the fact that changes may be made to objects by threads that are not executing transactions, all data items locked by an atomic transaction must be written to the object's secondary storage image. This is true regardless of whether the data is modified by this transaction or not (i.e., even data that is locked in read-mode must be committed). (Note that this requirement can be eliminated if transactions are used within the object. Then all threads will be executing transactions, whether or not they were executing one when the object was invoked.)

A major concern with the use of atomic transactions is the potential restriction that they place on the degree of concurrency that can be obtained from object implementations. Because distributed computer systems provide the opportunity to exploit the concurrency available in applications, any restriction on concurrency is undesirable. One of the major goals of Alpha is to permit the exploration of highly concurrent forms of atomic transactions. It has been shown that to increase the concurrency available with atomic transactions, semantic information about the actions being performed must be provided [Garcia 83]. This semantic information is difficult to automatically derive from programs (however the object model does contribute in this regard).

Some work has been done in the area of loosening the constraints of serializability. In particular, a form of atomic transaction has been proposed that offers a high degree of modularity and concurrency and promises to be practical in providing failure management and recovery within a real-time operating system [Sha 85]. The atomic transaction mechanisms in Alpha provide the means for the validation of these claims. Also, the decomposition of atomic transaction mechanisms allows the relaxation of some atomic transaction constraints. For example, the relaxation of the constraint of serializability can be accomplished by unlocking data items prior to committing the transactions in which they were locked. Additionally, to further exploit the potential concurrency in an application, compensating actions can be used in place of the traditional *roll-back* type of abort operations in objects.

In order to place upper bounds on the amount of time it takes for atomic transactions to commit or abort, the invocation mechanism in Alpha includes a means of autonomously detecting node failures and eliminating orphans. This is done by having each node keep track of the state of other nodes visited by the transaction, and if one of these nodes is found to have failed, the operations on the node involved in that invocation chain are terminated. This bounding of commit time is accomplished at the cost of increased communication overhead [McKendry 85].

4.3.2 Object Replication

In order to meet its goal of availability of services, the Alpha kernel provides support for the replication of objects. This support does not, in itself, constitute a complete data replication facility. The kernel provides a framework for experimentation in the area of replicated object management. The approach to object replication taken in Alpha is one based on the use of multiple instances of a particular type of object, all of which share a common logical identifier, and consequently appear as replicas of a single object. The general issue of object placement is considered a higher-level issue in Alpha, and therefore the question of how the replicas are to be distributed among physical nodes is addressed at a level above the kernel.

As with atomic transactions, the area of object replication is currently being explored as a part of an ongoing thesis project. The mechanisms currently provided by the kernel are meant to be representative of a much more comprehensive set of mechanisms to be developed as a result of this work.

The Alpha kernel is designed to be a framework to support a wide range of replication schemes, including *inclusive* and *exclusive* forms of replication. In the inclusive form of replication the replicas of an object function together as a single object. Therefore, an operation invoked on an inclusively replicated object has equivalent results to performing the operation on all of the existing replicas. The intent of this form of replication is to increase the availability of an object through a redundancy technique similar to an *available copies* replication scheme [Goodman 83].

To reduce the cost associated with such a replicated invocation, a *quorum* method may be used [Herlihy 86]. By associating a quorum with each operation defined on an object, fewer than the currently existing set of replicas can be involved in a particular operation. This mechanism can be used by the client to create objects with different degrees of availability, response time, and consistency.

A number of issues related to inclusive replication have been deferred in an effort to reduce the scope of this effort. For example, the timestamp or version number mechanisms needed to implement a proper quorum-based replication scheme have not been provided. Nor has the issue of the regeneration of failed replicas been addressed. These issues are being dealt with in a related thesis project.

For the exclusive form of replication, the kernel also provides a number of replicated instances of an object type. In this case, however, an invoked operation need only be performed on any one of the replicas for the operation to be considered complete. This approach provides a form of replication in which any one of a pool of undifferentiated object replicas is chosen, based on a global policy designed to meet certain goals. Examples of replica selection policies are “select the first replica that responds”, “select the replica that exists at the least loaded node”, and “select a replica near the invoking object.” The policy that selects the first replica that responds provides the potential for a higher degree of both availability and performance than is achievable with non-replicated objects. However, this is accomplished at the cost of not maintaining the consistency of data across the individual replicas.

4.3.2.1 Concepts

Replication provides a means to increase the availability of critical data and services. This increased availability can be used to allow data and services to survive node and communication failures. In addition, even when there are no failures, the time required to gain access to data or receive service can be reduced in certain situations by the presence of data and service replicas. In the latter case, a larger pool of servers should reduce the likelihood that all of them are busy when a service or datum is needed; furthermore, it may be possible to access a service locally incurring less expense than required for accessing it remotely.

The *replication factor*, that is, the number of replicas, for a given service or piece of data—along with the policies employed to manage the replicas—determines the robustness of the service or datum. Potentially, each additional replica can represent one more node or communication link failure that can be survived by the replicated service or data. A large replication factor can increase survivability, but this may be accomplished by trading away performance. This, along with a number of other relevant issues are discussed in the following sections.

4.3.2.2 Approach

In Alpha, all data and services are embodied by objects and therefore, their availability can be increased by replicating the object instances that encapsulate the desired services and data. In fact, the real requirement—and the requirement that Alpha fulfills—is that the desired services or data behave as if they were replicated. In the discussion that follows, a replicated object in Alpha should be thought of as a collection of objects that function as a single conceptual object. Often, these objects will be copies of one another.

As with all of the requirements that Alpha wishes to satisfy, requirements related to the availability of critical services and data are gathered from the application being supported. Therefore, an application's designers and implementers must determine the degree of availability required for a given service or datum. For example, an application for an N node system may require that a specific database must be available anytime the system is in operation. If that system can function with a minimum of M nodes, then a replication factor of N-M may be chosen to guarantee that, with reasonable replica placement, the database will be available on at least one node for every possible set of M or more surviving nodes. Based on such considerations, application designers and implementers determine which objects—representing critical services and data—must be replicated and, for each of these objects, what the replication factor should be to meet the application's availability requirements.

As with transactions, Alpha's kernel does not provide a complete object replication facility. Rather, it provides a set of mechanisms that have been designed to support a range of replicated object management policies. The policy may be provided by higher levels of the operating system or by the application directly. In either case, the mechanisms provided by the kernel are intended to provide necessary functions in an efficient manner. They provide the policy definer with necessary tools, while employing a unified approach to solve low-level problems in a manner that is compatible with other kernel mechanisms. They allow the policy definer to specify important behaviors that tailor the facility to the specific needs of the application. By using well-designed mechanisms, there is no need to worry about different high-level policies have disastrous conflicts at lower levels.

4.3.2.3 Mechanisms

The current object replication mechanisms are an initial attempt to provide an appropriate set of building blocks for replication. On-going work in the area of replication will allow later refinement of these mechanisms. The mechanisms presented are sufficient to satisfy the requirements of a straightforward object replication facility (in fact, that facility has been implemented) and are capable of satisfying the more ambitious requirements of other replication facilities that will be investigated in the future.

The mechanisms provided, in whole or in part, to support object replication are:

- **replicated object creation** — the ***ObjectManager*** provides an operation that will create a specified number of copies of a designated object type, all with the same initial state; the replicas are each placed on nodes according to a higher level placement policy;
- **identifier generation** — this is a general mechanism that has a specific application in support of replication: reserving a portion of the identifier name space for replicated object identifiers; therefore, an object can be recognized as replicated merely by looking at its identifier; replicated object identifiers also indicate the replication policies that are to be used for managing the object(s);
- **object/identifier binding** — once again, the binding of an identifier with a specific object is a general mechanism with a wide range of applications (such as well-known identifiers and the corresponding capabilities); this mechanism may be used to allow a replica to assume a second identifier (the identifier of the replicated object);
- **replica selection on invocation** — the operation invocation mechanism notes that an invocation is being done on a replicated object and follows the specified replica selection policy for that replicated object;
- **invocation completion handling** — at the completion of each operation performed on a replica, the invocation completion policy specified for that replicated object's replication scheme will be executed;
- **invocation failure handling** — if an operation invocation fails due to the loss of one or more replicas, the replication scheme's invocation failure policy — which possibly replaces failed replicas — will be executed and the invocation will optionally be re-initiated;
- **object checkpoint mechanism** — this mechanism can copy the modifications made to an object since the previous checkpoint operation to other designated replicas; this involves copying both data encapsulated by the object and semaphore and capability list modifications;
- **lost replica detection** — this mechanism detects the fact that a replica has been lost; once detected, the object replication policy's replica replacement policy will be executed.

4.3.2.4 Usage

The object replication mechanisms are intended to be able to provide a range of different object replication facilities when combined as prescribed by corresponding replication policies.

This section will first describe the use of these mechanisms in the initial, fairly straightforward, replication facility provided on Alpha and will conclude with a few general observations about the use of object replication under Alpha.

Under the initial replication scheme, any object can be replicated without the addition of special user code in the object to support replication. Instead, all of the support is provided by the operation invocation mechanism, the identifier generation mechanism, and the ***ObjectManager***. The replication scheme employs a number of replicas, one of which is designated as the master copy, while the others are designated as backup copies.

Each replicated object has a unique identifier, like every other object in the system. When the ***ObjectManager*** creates a replicated object, it actually creates a specified number of replicas, each of which has a unique identifier. In addition, the identifier generation mechanism issues another identifier that acts as the unique identifier for the replicated object. The replica on the same node as the ***ObjectManager*** is designated to be the master copy, and all of the other copies are designated as backup copies. The identifier for the replicated object is associated with the master copy by means of the object/identifier binding mechanism. As a result, all invocations on the replicated object (which, of course, use the replicated object identifier) are actually performed on the master copy. Then, the changes made by each operation execution are propagated to the backup copies by means of the checkpoint mechanism. This mechanism is employed at the completion of each operation on the master copy as part of the invocation completion policy for the replication scheme.

Should the master copy ever fail, the invocation failure handling mechanism executes the replication scheme's invocation failure policy. In this case, a simple procedure is employed to locate and promote one of the backup copies to be the new master. This promotion occurs by using the object/identifier binding mechanism to associate the unique replicated object identifier with the selected backup copy. Optionally, an additional backup copy can be created to replace the one that was just promoted to master copy.

Furthermore, threads are placed in the backup copies when they are created by the ***ObjectManager*** (on the master copy's node) in order to detect their failure. Should a backup copy ever disappear, the thread that was placed in it will break, and the thread will be trimmed—allowing the thread to execute code in the ***ObjectManager*** again. This code replaces the backup copy, if possible. Consequently, the ***ObjectManager*** makes sure that there are always backup copies available in case the master copy should fail.

Many other replication facilities can be built with the mechanisms presented in the previous section. For example, the unique name assigned to the replicated object could be used by several copies simultaneously. Or, it could be used by all of them at once, in which case the replica selection mechanism could be used to determine the subset that should perform any given operation invocation. Such an approach could be used to implement quorum-based replication schemes. Replication facilities that require greater concurrency may not use the checkpoint mechanism as often as the scheme outlined above. In fact, if the replicas did not need to cooperate or maintain some form of mutual consistency (say if they provided access to a service or a large, read-only database) then all of the replicas could share the same replicated object identifier and the replica selection mechanism could just pick the least busy replica whenever an invocation was performed. Checkpoints would never be needed, and so would not limit the available concurrency for the object.

The following section mentions some of the issues that must be addressed when designing and implementing replicated object facilities.

Finally, notice that the use of certain object replication facilities in concert with transactions can potentially provide a powerful, high performance building block. Transactions can be used in accessing the data within a replica to guarantee a consistent view of data between separate threads. However, there is no need to have a secondary storage image of the replica in some form of stable storage, as would normally be required. If the replica is lost, there are other replicas to replace it, and they have the necessary state available to them (without needing to access stable storage). The trade-off that must be evaluated is: whether the coordination required among replicas is less expensive than the overhead imposed by storing all secondary storage images in stable storage. If so, then it is probably desirable to use replication as described above. In fact, if it has already been determined that the object should be replicated for purposes of increased availability, then it would seem very likely that this approach would have merit.

4.3.2.5 Issues

There are a number of issues that involve replication. The following discussion deals with some of the most critical issues addressed to date.

First of all, there is the basic question of what to replicate. Once again, Alpha clearly should replicate objects because that is where all of the data are encapsulated. But should object replicas be created and then have every operation invocation be executed on each of the replicas (possibly in parallel)? For the initial replication policy, we do not believe that it should. This is because the real issue being addressed is the replication of state, and state can be copied without having to re-execute operations on each object replica.

Using threads to execute an operation in parallel on each of the object replicas is a poor solution for a number of reasons:

- Alpha's scheduling policy is quite complex, and it may be impossible to guarantee that the same operations could be scheduled to execute on each object replica; these replicas are probably located on different nodes with different instantaneous workloads (remember, too, that the scheduling parameters for computations can vary as a function of time, further complicating matters);
- even if the same operations could be scheduled, they would have to execute in the same order; for example, if on one replica a given thread acquired a semaphore before some other threads, it must acquire the same semaphore in all of the replicas or there will be a danger of inconsistency and even deadlock;
- if the threads do not complete the operation on their replicas at approximately the same time, progress may be halted until the slowest thread finally completes the operation; and
- to actually execute the operation requires the use of the AP (Application Processors), thereby causing APs across the system to perform duplicate work; instead, a single AP could perform the work and then a number of CPs (Communication Coprocessors) and SSPs (Secondary Storage Coprocessors) could propagate the necessary state to other copies without further intervention by any APs.

Also, some other semantic and performance issues arise when executing an operation on multiple replicas, rather than one. For example, imagine that an operation is invoked on a replicated object and that the replicated object invokes another object while in the process of executing the operation. If the new object is also replicated, then even more threads have to be created (or forked) to handle the new invocation, at an even greater cost in terms of cycles consumed to carry out the operation. Of course, the situation can continue, with replicated object invoking operation on replicated object, with an exponential explosion of threads to perform the operation invocations.

On the other hand, if a replicated object invokes an operation on a non-replicated object, what is the proper behavior? It would seem that the operation on the non-replicated object should only be performed once, but it will be performed once for each replica if no special precautions are taken. Furthermore, the necessary precautions would seem to require a good deal of coordination among the replicas or a great amount of semantic awareness on the part of the non-replicated object. Both of these alternatives are discouraging.

The preceding discussion clearly argues for the approach taken initially: replicate the desired state and propagate state changes. The semantics are natural, performance is superior to the alternatives just mentioned, and the necessary functions to provide availability are supplied.

A second issue that arose from the consideration of replication was the object programmer's point of view. Should the programmer have to write an object differently if it is to be replicated than if it will not be? The initial replication scheme allows any object to be replicated. No special objects must be written, thereby simplifying matters for the programmer and providing the greatest amount of flexibility for the application.

Since the solution employed in the first replication facility is general purpose (i.e., it can be used on any object) it has not been tailored for any specific object. If the programmer were aware of replication, then object-specific optimizations could be performed that take advantage of the semantics of the replicated object. The degree to which such optimizations can be exploited remains to be seen.

In Alpha's first replication scheme, all of the replicas are the same (at least at the completion of each checkpoint execution). This is not true for all replication schemes. For example, in quorum replication schemes, a group (a quorum) of replicas must jointly cooperate to provide any service. Among them they have all of the information required, although no two may be identical. The object programmer may have to dictate the manner of coordination among the replicas under such a scheme. To minimize the programmer involvement as much as possible, an appropriate framework must be present that facilitates the development of object definitions for quorum schemes, and supports their execution. The mechanisms in place are intended to provide this framework for further work in this area. Additionally, software libraries may be developed for programmers to use when defining object types.

Initially, the policies that define the replicated object facility have been provided by the system, not the application programmer. The application programmer must be permitted to select the replication facility to be used for each replicated object created, but has not been permitted to provide the definition for that facility at run-time. This is similar to the approach taken by Alpha with respect to scheduling policies: there are a number of policies available that are intended to span the needs of the application, and the application is able to select the policy that best suits its needs. Of course, as any genuinely new needs are demonstrated by Alpha applications, the available set of replication policies can be expanded accordingly.

The placement of replicas should be mentioned. Since replicas are often employed so that data and services will survive node and communication failures, it will often be the case that placing multiple replicas of a given object on a single node makes no sense. If the node were to fail, multiple replicas would be lost. So having more than one replica there did not make the data and services more survivable. Even when replicas are present only to provide duplicate, independent services, it may not make sense to place two of them on a single node (because, unless the node has multiple application processors, the opportunities to achieve actual concurrency will be limited). However, depending on the concurrency restrictions imposed by an object, multiple copies (even on a single node) may improve responsiveness.

In general, the replication factor or the size of a quorum impacts performance. The number of copies that must be involved to carry out an operation will impose a cost in terms of resources. Different replication schemes may spread the resource cost among application processors, communication processors, scheduling processors, and storage management processors differently, but in every case, a price will be paid for each copy that must be accessed. This should not be distressing since it is simply the price that must be paid for increased availability. Of course, because the cost imposed by different schemes will vary, care should be taken to minimize the price that is paid for critical applications. Alpha's ability to support a number of replication schemes (simultaneously, if necessary) facilitates the development of appropriate schemes.

The initial replication scheme provided by Alpha restricts concurrency since each replicated object can only allow one thread at a time to modify its data. All other threads that wish to modify the object's encapsulated state are blocked until the checkpoint operation that is initiated at the conclusion of the writing thread's operation completes. The effect of this restriction is that all of the replicas have operations serialized on them, and so, in some sense, represent consistent snapshots of the replicated object's state. Notice that this restriction is not as great as it might seem since threads often have a significant amount of state stored on their respective stacks. The stack data reflects thread-specific data, while the data encapsulated by the object represents global data shared by all of the threads in the object.

There is other global object state that is shared by all of the threads in an object, and so must be replicated. Specifically, the capabilities held by the object and the state of its semaphores constitute global object state that must be replicated. The checkpoint mechanism currently implemented copies all of the required object state to all of the specified replicas reliably. Any other replication schemes adopted in the future will also have to be concerned with *all* of the state of an object.

Replicated objects are provided to ensure that the replicated data and services survive node and communication failures. Notice that this is a statement about objects and object state. Following a node failure, any threads that passed through or were executing in any replicas on the failed node are broken and must be trimmed. Hence, they will resume execution either in some object other than the replica in which they had been, or they will resume execution in the invocation mechanism and will execute the replication scheme's invocation failure handling policy. In any case, the thread sections that resided at least partially within the replicas will be lost. Another approach to replication might investigate support for stable threads as well. (Stable threads would be threads that survive node failures intact. That is the sections of the threads that would otherwise be lost would be reconstituted elsewhere without loss of any state.) This would presumably involve replicating a thread's state (including all of its stack) and *frequently* noting its current position in an object. Consequently, it would be quite expensive and might not perform very well.

One factor that must be emphasized in future replication policies is time. As with a number of other areas (e.g., scheduling, transactions, and communications), Alpha can draw on a large body of existing research and technology to identify promising ideas and approaches. However, the project's view of time constraints affects all aspects of Alpha, and replication will be no different. Highly concurrent schemes may be generally desirable, but scheduling multiple threads to carry out an invocation on a replicated object, or providing services according to the general notion of best-effort decision-making embodied by Alpha will present some difficulties and will supply significant constraints on which schemes are appropriate.

Finally, one more potential direction for further work can be mentioned. Currently, the application programmer must determine the replication factor and the replication scheme for each replicated object created. Some interesting work may be done to allow the operating system to determine the appropriate replication factor and replication scheme for an object based on a specification of object reliability and responsiveness that is more meaningful to the application programmer.

5 Comparisons with Other Models

In order to illustrate some of the more significant aspects of the Alpha programming model, this chapter compares selected features of Alpha with other, more common programming techniques. In addition, key differences between traditional programming models and that supported by the Alpha operating system are highlighted.

5.1 Conventional Approaches

The following subsections attempt to illuminate some of the salient features of the Alpha programming model by way of comparisons with more familiar concepts. The goal here, while attempting to be fair and accurate, is to emphasize the major differences that exist between the various programming models. Towards that end, some liberty is taken in defining the representative behavior of the models.

For comparison with the features of Alpha, the traditional process/message-based approach is examined, as is the client/server model of process interaction.

5.1.1 Process/Message Approach

In most systems, application functions are decomposed into some form of software module for reasons of intellectual manageability, reusability, relocatability, and concurrency. In most modern operating systems the software modules that are combined to make up computations are known as *processes*. While the exact definition of a process varies from system to system, most conform to the original notions described in [Hansen 70] and [Dennis 66]. In addition, the notion of *message passing* is frequently coupled with processes to create what is known as process/message-based system. In such systems, the computations that comprise an application are mapped onto collections of processes that interact via the exchange of messages. The exact semantics of message passing facilities vary even more widely than those of processes, but, again, most conform to a common set of basic principles—a unit of data is passed from one process to another via the explicit execution of send and receive commands. An example of a representative process/message system is shown in Figure 16.

The traditional notion of a process is equivalent to an Alpha object and a "captive" thread (i.e., a thread that remains within a single object throughout its existence). Threads and objects in Alpha represent the cleaving of traditional processes into a locus of execution control, and the code and data that make up the process. From an implementation standpoint, a snapshot of a single thread executing within an object is indistinguishable from a snapshot of an executing process (i.e., in both cases there is a point of control executing within a module containing both code and data).

A process, by definition, performs a single activity, and additional processes are created (and their execution started) in order to achieve concurrency. In Alpha, concurrency of execution is achieved by having multiple threads execute concurrently. In comparison to typical processes, threads are "light-weight" entities—i.e., have minimal state associated with them and can be created and deleted quickly. Processes are typically "heavy-weight" programming structures, and are therefore favor large-grained concurrent structures. In order to support finer-grained concurrency, some form of process substructure (e.g., light-weight processes) is frequently introduced in process/message systems. Commonly, this involves the creation of units of execution that execute concurrently within a common process's address space. Equally commonly, such substructure is implemented above the system interface. The consequent loss of correspondence between user and system abstractions can result in increased programming effort and sub-optimal system resource management (this point will be elaborated upon in the following section). Furthermore, the unconstrained sharing of information within a process sacrifices both modularity and fault containment—e.g., sub-processes must be co-located on the node with the process within which they are executing, and failures (like stack overflows) in one sub-process can affect other sub-processes. With the Alpha programming model, a high degree of concurrency can be achieved without compromising the modularity provided by the encapsulation of data within objects. This is because a potentially unlimited number of threads can execute within a single object, and the information encapsulated by an object cannot be accessed via any other object (i.e., there is no sharing of memory among objects).

Furthermore, concurrency in Alpha is achieved without the use of asynchronous communications—each concurrent activity is carried out by a separate thread, and threads do not perform *fork* and *join* operations; instead new threads are created and destroyed. This allows the more familiar, procedure-call-like form of communications to be used without the penalties typically associated with synchronous communications facilities.

In general, processes do not have well-defined interfaces—the entry points into a process (as defined by message receive commands) can exist at arbitrary places within the code of a process. Conversely, an object in Alpha has a rigid, well-defined interface, as specified by the operations defined on the object. This difference results in modularity benefits similar to those obtained by co-locating synchronization constructs in a monitor-like structure as opposed to having semaphore operations dispersed among the processes in an application. Objects also provide a formalized, system-defined and recognized interface that permits the separation of an object's specification from its implementation. In order to achieve similar benefits, some systems implement object-like structures on top of a native process/message system interface (e.g., Matchmaker [Jones 84]).

It is also frequently the case with process/message systems that the distinction between local and remote processes is made explicit to the programmer. This means that the client must manage the mapping of logical identifiers to physical locations. In Alpha, the effects of physical distribution are made totally transparent to the programmer and the programmer is not burdened with the (recurring) cost of dealing with physical communications.

5.1.2 Client/Server Approach

A popular convention for structuring the use of processes and messages is known as the *client/server* programming model. In this approach, applications consist of processes which provide a given service (i.e., *servers*), and processes which use the provided services (i.e., *clients*). Client processes send messages to server processes in order to obtain a service, and server processes return a message when the service has been provided. Any particular process can be a client, a server, or both a client and a server.

In Alpha, the analog to the sending of messages to a server process is the invocation of an operation on an object. The major differences between the Alpha programming model and a typical client/server model implementation are: server processes do not present well-defined interfaces to clients, server processes limit the ability to exploit an application's potential for concurrency, and the client/server model does not maintain correspondence between the programmer's logical view of computations and the system's implementation. The first two of these differences stem from issues similar to those discussed for process/message systems. The fact that the client/server model introduces a discontinuity between the application programmer's and the system's representation of computations is responsible for a highly significant deficiency in the model.

In general software engineering practice, applications are decomposed into programming modules (e.g., processes or objects), which are interconnected via a system's inter-module communications facility (e.g., messages or operation invocations). Each of the concurrent activities that make up an application can be thought of as independent computations, consisting of a sequence of interconnected activities. In a client/server system, each "step" of a computation is initiated by sending a message to a server, and results in the execution of a server process, the completion of which is indicated by a message sent from the server back to the client. An application's logical computations are not visible at the system level, but rather, the system is only aware of a collection of (seemingly unrelated) processes. In effect, the application programmer's view of the individual computations, which defines the interrelationships between client and server processes, is not available to the system.

In contrast to the client/server approach, applications for Alpha are decomposed into objects, and each computation is implemented by a thread that moves among objects via operation invocations. The thread and object programming model supported by Alpha permits implementation structures quite similar to the client/server model. For example, objects can be used to implement both clients and servers, and threads executing within client objects perform invocations on the operations offered by server objects to obtain the desired services. While these two approaches have many features in common, the most significant difference is that the Alpha programming model provides a direct correspondence between the abstraction and implementation of computations. Threads are not bound to a specific object, but move among objects (and across nodes) in order to perform the various steps of a computation. Threads provide the system with an entity that performs a separate activity, corresponding directly to the application programmer's logical notion of a computation. Therefore, the system can associate requests for system resources with a computation that has application-level constraints and attributes associated with it, permitting the system to more effectively perform its resource management functions. Whereas the execution of a thread is performing a single application-level computation, a process in the client/server model executes on behalf of many such logical computations. Furthermore, a thread carries its application-provided attribute information along with it as it executes within different objects, thereby providing the continuity for a logical computation that is missing in process-based client/server implementations.

The system-level visibility of logical computations provided by the Alpha programming model has a number of benefits over common client/server models, including the fact that the attributes of a computation (e.g., importance and urgency) are not lost as each step is executed. This is not the case with the client/server model, where a step of a computation is performed at the priority of the server, not the priority of the client (representing the previous stage of the logical computation). Also, with Alpha, the system's resources are managed, according to a specific policy, on the basis of application-specified, per-computation attributes that reflect the needs of the application, and are expressed in terms of application-level abstractions (i.e., the programmer need not transform the application's requirements into some arbitrary form, unrelated to the application's needs in order to cause the system to behave in the desired fashion). This approach allows the application-defined requirements to dominate the effects of the communication and scheduling subsystems (along with their interactions) on the system's behavior. In addition, Alpha threads do not require interaction with the scheduling subsystem or a complete context swap for each step of a computation. The following section provides an extended example of this point. Finally, while it may be possible to emulate or approximate various features of the Alpha programming model with other programming models, Alpha represents a more-or-less ideal manifestation of the principles essential to provide effective system support for real-time applications.

5.2 An Example and Issues/Implications

To illustrate some of the significant differences between Alpha and traditional client/server-based systems, consider a simplified application consisting of three separate computations, each of which has a portion in common with the others (i.e., they make use of some common service, or they perform the same sequence of instructions). Figure 16 illustrates a stylized client/server implementation of this example application, where processes 1, 2, and 3 obtain the service provided by process 4 through the exchange of messages. Figure 17 provides an illustration of the same application implemented in Alpha, where threads *A*, *B*, and *C* begin execution in objects *i*, *j*, and *k* (respectively), and obtain the service provided by object *x* by invoking operations on it.

A number of significant differences exist between these two approaches. For example, in the process/message case there is a mismatch between the programmer's logical concept of *three* application computations being carried out by *four* interacting processes in the implementation. This discontinuity requires that the programmer transform the application's conceptual design into a differently structured realization. (While the transformation is not significant in this example, it can be quite significant for less trivial applications).

Furthermore, the discontinuity between the application programmer's and the system's views of computations interferes with the system's ability to perform global time-driven resource management. To illustrate this point, note that in Figure 16, each computation is being performed by the cooperative interaction among a pair of (as far as the system is concerned) independent processes, and, as is typical for such systems, each process is scheduled based on its own, static priority. In this case, the system manages resources based only on the requests it receives from individual processes, and not based on the characteristics or requirements of the logical computations. This results in the server process executing at its given priority level, regardless of on whose behalf it is functioning. So the portion of the first computation being provided by process 4 is executed at too low a level of priority, while part of the third computation runs at a higher priority level than it should.

Process	Priority
1	High
2	Medium
3	Low
4	Medium

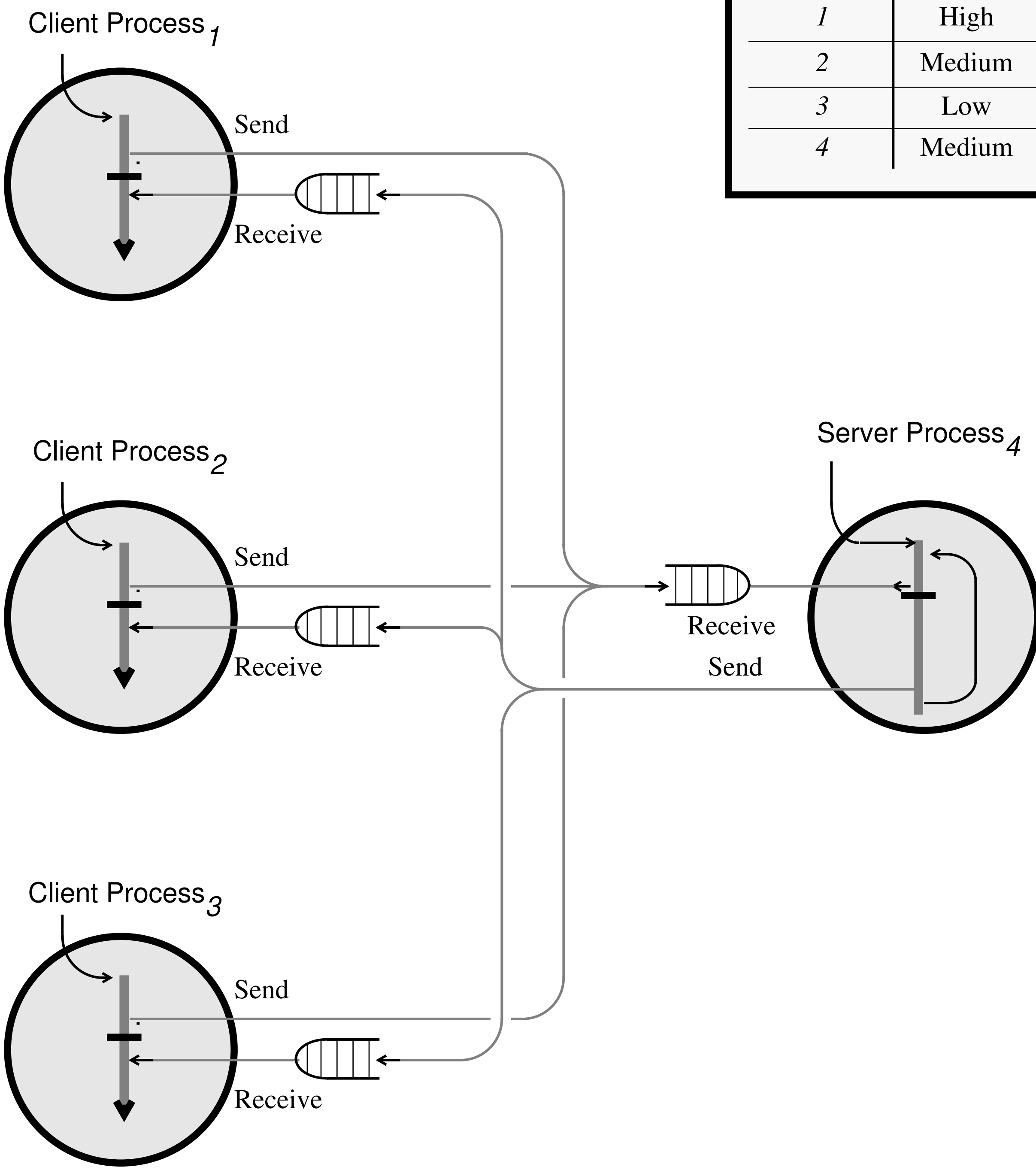

Figure 16: Typical Process/Message Interactions

In Alpha, each thread represents an individual computation and moves among nodes freely, carrying with it the attributes assigned to the computation by the application programmer. This approach maps the logical computation directly onto its system manifestation and allows the system to receive resource requests from the computations themselves and not some unrelated artifact. The result of this is that each of the threads in Figure 17 execute in their proper order (based on their application-specified urgency and importance attributes), regardless of the object they happen to be executing in at any given time. In fact, the server's function is performed, not only with the attributes of the individual computation requesting the service, but actually by the computation itself.

An additional benefit of the Alpha thread/object abstractions is that they admit of an implementation that does not involve unnecessary scheduler interactions on each instance of inter-module (i.e., inter-object) communication. In typical process/message-based systems, communication and scheduling activities are intertwined in an undesirable fashion—to accomplish the next (logical) phase of a computation requires a scheduling activity. This is an artifact of the process/message approach, because communication itself does not require a scheduling event. When the system's scheduler has determined that a computation should receive processor cycles, the computation should retain the processor until a legitimate scheduling event occurs, and not relinquish the processor whenever it wishes to execute another part of the computation. This problem is intrinsic to these styles of programming—in the Alpha version of the example, there is a single schedulable entity for each computation (i.e., a thread for each computation), while in the process/message example there are multiple schedulable entities involved (i.e., a pair of processes for each computation).

If a computation is thought of as a locus of control that makes use of the operations provided by various objects in performing its function, then each invocation of an operation on an object can be thought of as indicating the next step of the computation. Given such a view, there is no logical reason why a computation should be required to interact with a system's scheduling facility before taking each step. Once the scheduling facility has bound a computation to a processor, it should only be unbound when a scheduling event occurs, and the movement of a thread from one step to another does not constitute an interesting scheduling event. Clearly, if the scheduling facility must be involved in each processing step, unnecessary system overhead is incurred and it becomes more difficult to ensure that timeliness guarantees associated with the client-level computation can be met. With an object- and thread-based approach, a thread moves among its steps without involving the scheduler, until a scheduling event occurs (e.g., a time quantum is exhausted, a higher priority thread is available, or an operation was invoked that blocks the thread). It should be noted that, should a thread cross a node boundary as a result of an invocation, a scheduling decision must be made because the thread must contend for the processor with the other threads at the destination node.

While the emphasis in this example was on the management of processor cycles (i.e., scheduling), the Alpha programming abstractions allow for similar time-driven management of any I/O, communication, or memory resources required by computations. In most process/message-based systems the management of these resources interact in an arbitrary (and largely uncontrollable) fashion. Alpha threads provide a unifying means of managing system resources in a consistent fashion both within and among the system's nodes.

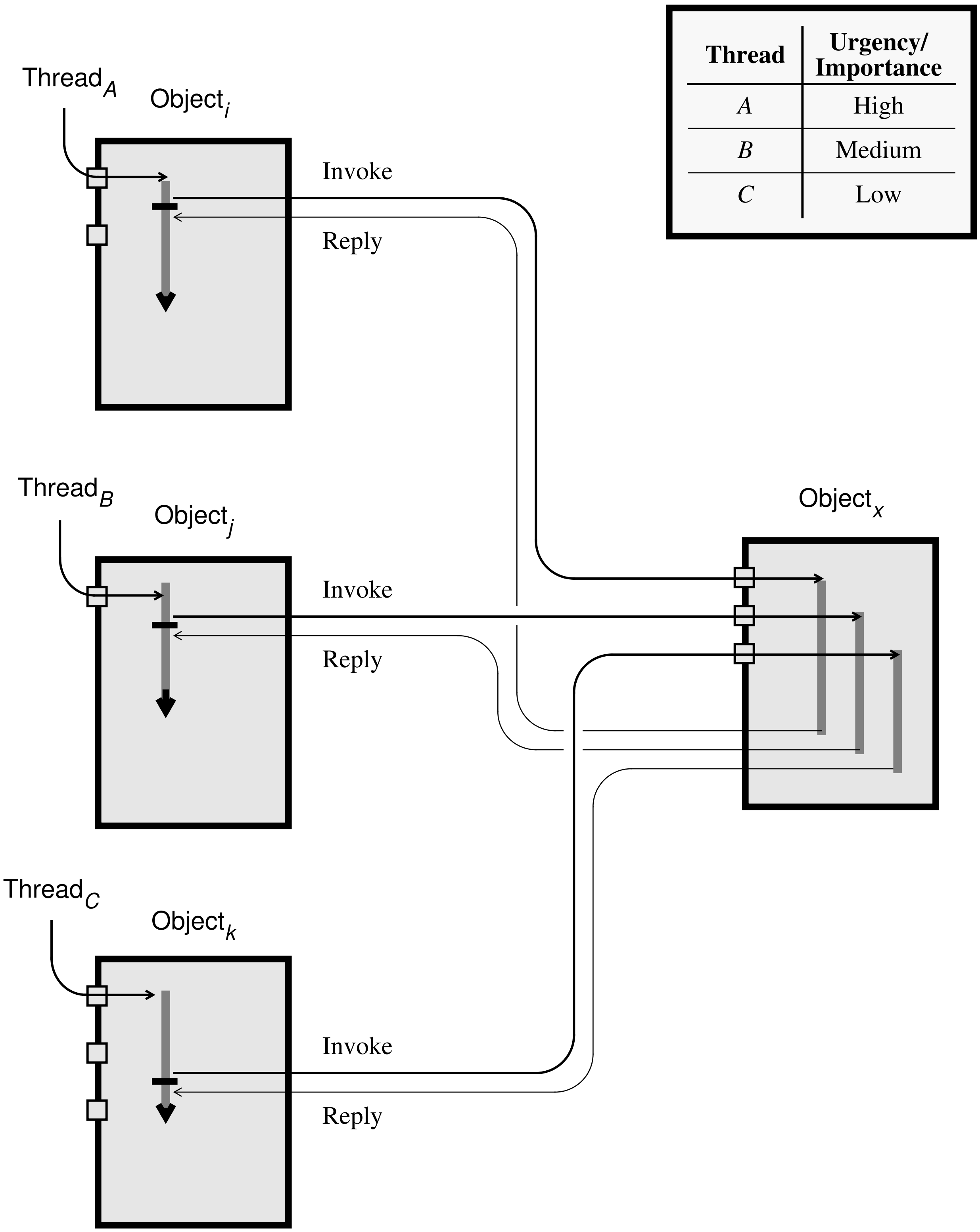

Figure 17: Typical Thread/Object Interactions

In some message-based systems, the priority of a message can be used to resolve conflicts for communication resources. To resolve contention for all types of resources, computations would have to retain their priority across all of their steps. To do this would require that server processes assume the priority of each message they receive. Furthermore, the server processes must be preemptable so that when a message with higher priority than the current one arrives, the server must suspend its current work to allow new work to begin on behalf of the newly received message. It is (at best) a difficult proposition to so intertwine the scheduling and communication facilities of a system. Thus the thread/object approach taken in Alpha seems a much more promising approach to effectively resolving the contention for resources introduced by distributed real-time tasks.

6 Acknowledgments

Many people contributed to this effort, and many others continue to contribute as the Archons project enters its next phase.

Doug Jensen is the founder of the Archons project. He obtained the support for the project over its 10-year existence, and provided most of the fundamental philosophy and concepts for Alpha. Doug continues to provide support and guidance for this research effort as Release 2 of the Alpha operating system is being developed at Kendall Square Research Corporation (KSR).

In his brief visit with the Archons project, Martin McKendry initiated the implementation effort that has become Alpha and provided many of the initial implementation concepts.

Sam Shipman worked on various subsystems and support tools, and contributed to refining and completing the system design and implementation. David Maynard also worked on various aspects of the system's design and implementation, and acted as the project liaison with General Dynamics during our joint C^2 demonstration effort. David is now working on a thesis in the area of real-time scheduling for decentralized computers with multiprocessor nodes. Huay-Yong Wang worked on the scheduling subsystem and the design and implementation of various other system functions.

While this report describes the programming model for Release 1.0 of Alpha (being implemented at CMU), it is not totally a historical document, but rather, reflects the current thinking on Alpha, which has been significantly influenced by the efforts of the Alpha team at KSR. The KSR team includes Ed Burke, Jim Hanko, Franklin Reynolds, and Jack Test. All of these people have contributed in one way or another to the work described here, and are busily working to supersede the current version of Alpha with a much enhanced, production quality, Alpha operating system.

Other project members that contributed to this effort are Jeff Trull, Chuck Kollar, Bruce Taylor, Don Lindsay, and Dan Reiner. Thanks are due to Tom Lawrence and Dick Metzger, the Archons project's prime sponsors at the Rome Air Development Center. Additionally, we would like to thank Russell Kegley and Calvin Head of the Fort Worth Division of General Dynamics Corporation for their assistance in our joint C^2 application development effort.

References

[Ada 83] United States Department of Defense.
Reference Manual for the Ada Programming Language.
ANSI/MIL-STD-1815A-1983.
Springer-Verlag, New York, 1983.

[Allchin 83] Allchin, J. E.
Support for Objects and Actions in Clouds.
School of Information and Computer Science, Project Report, Georgia Institute of Technology, May, 1983.

[Almes 85] Almes, G. T., Black, A. P., Lazowska, E. D. and Noe, J. D.
The Eden System: A Technical Review.
IEEE Transactions on Software Engineering SE-11(1):43-58, January, 1985.

[Anderson 81] Anderson, T. and Lee, P. A.
Fault Tolerance: Principles and Practice.
Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Baskett 77] Baskett, F., Howard, J. H. and Montague, J. T.
Task Communications in DEMOS.
Operating Systems Review 11(5):23-32, November, 1977.

[Bayer 79] Bayer, R., Graham, R. M. and Seegmueller, G. (editors).
Lecture Notes in Computer Science*,* Volume 60: *Operating Systems: An Advanced Course.*
Springer-Verlag, Berlin, West Germany, 1979.

[Berstis 80] Berstis, V.
Security and Protection of Data in the IBM System/38.
In *Proceedings of the Seventh Symposium on Computer Architecture*, pages 245-252, IEEE, May, 1980.

[Boebert 78] Boebert, W. E.
Concepts and Facilities of the HXDP Executive.
Technical Report 78SRC21, Honeywell Systems & Research Center, March, 1978.

[Campbell 74] Campbell, R. H. and Habermann, A. N.
Lecture Notes in Computer Science. Volume No. 16*.* Springer-Verlag, Berlin, 1974.

[Clark 88a] Clark, R. K., Kegley, R. B., Keleher, P. J., Maynard, D. P., Northcutt, J. D., Shipman, S. E. and Zimmerman, B. A.
An Example Real-Time Command and Control Application.
Archons Project Technical Report #88032, Department of Computer Science, Carnegie-Mellon University, March, 1988.

[Clark 88b] Clark, R. K.
Operating System Kernel Support for Real-Time Atomic Transactions.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University.
In progress.

[Cox 86] Cox, B. J.
Object-Oriented Programming.
Addison-Wesley, Reading, Massachusetts, 1986.

[Dennis 66] Dennis, J. B. and Van Horn, E. C.
Programming Semantics for Microprogrammed Computation.
Communications of the ACM 9(3):143-155, March, 1966.

[Eswaren 76] Eswaren, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

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

[Garcia 83] Garcia-Molina, H.
Using Semantic Knowledge for Transaction Processing in a Distributed Database.
ACM Transactions on Database Systems 8(2), June, 1983.

[Goldberg 83] Goldberg, A. and Robson, D.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, Reading, Massachusetts, 1983.

[Habermann 76] Habermann, A. N., Flon, L. and Cooprider, L.
Modularization and Hierarchy in a Family of Operating Systems.
Communications of the ACM 19(5):266-272, May, 1976.

[Hansen 70] Brinch Hansen, P.
The Nucleus of a Multiprogramming System.
Communications of the ACM 13(4):238-250, April, 1970.

[Hoare 74] Hoare, C. A. R.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):549-557, October, 1974.

[Jensen 76a] Jensen, E. D. and Anderson, G. A.
Feasibility Demonstration of Distributed Processing for Small Ships Command and Control Systems.
Final Report N00123-74-C-0891, Honeywell Systems & Research Center, August, 1976.

[Jensen 76b] Jensen, E. D.

The Implications of Physical Dispersal on Operating Systems.
Workshop on Distributed Processing, Brown University, Providence, Rhode Island, August, 1976.

[Jones 84] Jones, M. B., Rashid, R. F. and Thompson, M. R.
Matchmaker: An Interface Specification Language for Distributed Processing.
Technical Report CMU-CS-84-161, Department of Computer Science, Carnegie-Mellon University, 1984.

[Jones 79] Jones, A., Chansler, R., Durham, I., Schwans, K. and Vegdahl, S.
StarOS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings, Seventh Symposium on Operating System Principles*, pages 117-127. ACM, December, 1979.

[Kahn 81] Kahn, K. C., Corwin, W. M., Dennis, T. D., Hooge, H. D., Hubka, D. E. and Hutchins, L. A.
iMAX: A Multiprocessing Operating System for an Object-Based Computer.
In *Proceedings, Eighth Symposium on Operating System Principles,* pages 127-136, ACM, December, 1981.

[Lampson 69] Lampson, B. W.
Dynamic Protection Structures.
In *Proceedings of the Fall Joint Computer Conference,* pages 27-38, IFIPS, 1969.

[Lampson 76] Lampson, B. W. and Sturgis, H. E.
Reflections on an Operating System Design.
Communications of the ACM 19(5):251-256, May, 1976.

[Lampson 81] Lampson, B. W., Paul, M. and Siegert, H. J. (editors).
Lecture Notes in Computer Science. Volume 105: *Distributed Systems—Architecture and Implementation.*
Springer-Verlag, Berlin, 1981.

[Levin 77] Levin, R.
Program Structures for Exceptional Condition Handling.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, June, 1977.

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

[Liskov 84] Liskov, B. H.
Overview of the Argus Language and System.
Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, February, 1984.

[Liskov 85] Liskov, B. H., Herlihy, M. P. and Gilbert, L.
Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing.
Technical Report CMU-CS-85-168, Department of Computer Science, Carnegie-Mellon University, October, 1985.

[Lister 77] Lister, A.
The Problem of Nested Monitor Calls.
ACM Operating System Review 11(2):5-7, July, 1977.

[Maynard 88] Maynard, D. P.
Multiprocessor Scheduling for Real-Time Applications.
Ph.D. Thesis Proposal, Department of Electrical and Computer Engineering, Carnegie-Mellon University.
In Progress.

[McKendry 84a] McKendry, M. S.
The Clouds Project: Reliable Operating Systems for Multicomputers.
Project Report, Georgia Institute of Technology, 1984.

[McKendry 84b] McKendry, M. S.
Ordering Actions for Visibility.
Project Report, Georgia Institute of Technology, 1984.

[McKendry 85] McKendry, M. S. and Herlihy, M. P.
Time-Driven Orphan Elimination.
Technical Report CMU-CS-85-138, Department of Computer Science, Carnegie-Mellon University, 1985.

[Moss 85] Moss, J. E. B.
Nested Transactions: An Approach to Reliable Distributed Computing.
The MIT Press, Cambridge, Massachusetts, 1985.

[Nelson 81] Nelson, B. J.
Remote Procedure Call.
Ph.D. Thesis. Department of Computer Science, Carnegie-Mellon University, May, 1981.

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Requirements and Rationale.
Archons Project Technical Report #88011, Department of Computer Science, Carnegie-Mellon University, January, 1988.

[Northcutt 88b] Northcutt, J. D.
The Alpha Distributed Computer System Testbed.
Archons Project Technical Report #88033, Department of Computer Science, Carnegie-Mellon University, March, 1988.

[Northcutt 88c] Northcutt, J. D., Clark, R. K., Shipman, S. E. and Lindsay, D. C.
The Alpha Operating System: System/Subsystem Specification.
Archons Project Technical Report #88051, Department of Computer Science, Carnegie-Mellon University, May, 1988.

[Northcutt 88d] Northcutt, J. D.
The Alpha Operating System: Kernel Interface.
Archons Project Technical Report #88061, Department of Computer Science, Carnegie-Mellon University, June, 1988.

[Randell 78] Randell, B., Lee, P. A. and Treleaven, P. C.
Reliability Issues in Computing System Design.
Computing Surveys 10(2):123-165, June, 1978.

[Rashid 81] Rashid, R. F. and Robertson, G. G.
Accent: A Communications Oriented Network Operating System Kernel.
Technical Report CMU-CS-81-123, Department of Computer Science, Carnegie-Mellon University, April, 1981.

[Sha 85] Sha, L.
Modular Concurrency Control and Failure Recovery—Consistency, Correctness and Optimality.
Ph.D. Thesis, Department of Electrical Engineering, Carnegie-Mellon University, 1985.

[Shantz 85] Shantz, R., Schroder, M., Barrow, M., Bono, G., Dean, M., Gurwitz, R., Lebowitz, K. and Sands, R.
CRONUS, A Distributed Operating System: Interim Technical Report No. 5.
Technical Report 5991, Bolt, Beranek and Newman, June, 1985.

[Shipman 88] Shipman, S. E.
The Alpha Operating System: Programming Language Support.
Archons Project Technical Report #88042, Department of Computer Science, Carnegie-Mellon University, April, 1988.

[Sollins 81] Sollins, K. R.
The TFTP Protocol (Revision 2).
RFP-783, Network Working Group, MIT, June, 1981.

[Wilkes 79] Wilkes, M. V. and Needham R. M.
The Cambridge CAP Computer and its Operating System.
North Holland, New York, 1979.

[Wulf 81] Wulf, W. A., Levin, R. and Harbison, S. P.
Hydra/C.mmp: An Experimental Computer System.
McGraw/Hill, New York, 1981.

Appendix I: Programming Example

The following example object is taken from the joint Archons Project/General Dynamics C^2 application effort that demonstrated a reliable, distributed air-defense application implemented directly on the kernel of the Alpha operating system. The simple programming tools developed as a part of the operating system implementation effort made direct use of the kernel's native constructs and were used to create this sample application. Details concerning the programming language extensions used to access the kernel's functions, and the programming environment used to create the prototype application, are described in [Shipman 88] and [Northcutt 88d].

The object shown here is known as the ***TrackHandler*** object, and is representative of the other objects in the application. The ***TrackHandler*** object functions more as a director of threads' actions, as opposed to the more typical function of encapsulating data. The ***TrackHandler*** object oversees the processing of tracks throughout the application, and handles the cases where a previously uncorrelated plot is promoted to "real track" status, a track has been determined to be hostile (and must therefore have a thread created to deal with it), and a node has failed (and therefore lost data and activities must be recovered).

A detailed discussion of the C^2 application effort is given in [Clark 88a].

```
/*******************************************************************
 *                                                                 *
 *                   Distributed C2 Demonstrator                   *
 *                                                                 *
 *               General Dynamics Fort Worth Division              *
 *            Carnegie-Mellon University Archone Project           *
 *                                                                 *
 *******************************************************************/

OBJECT TrkHandlerType {

/***********************************************************************\
*
*  Track Handler:
*       directs the processing of tracks throughout the system.
*
* OPERATIONS:
* -----------
*    Initialize - Initializes the TrackHandler.  Currently just accepts the
*        CAPAs that are needed for processing.
*
*    ProcessTrack - Handles the processing of a new track that has entered
*        the system.  New tracks are identified, and if , handed off
*        to the threat handler.  Since we don't want to lose this activity
*        if a node fails, this activity is registered with the TrackDBMgr.
*
*    ProcessThreat - Determines the correct reaction to a threat that for
*        some reason needs to be reconsidered.  This may happen when the
*        threat is first identified, or when a previous intercept attempt
*        has failed.  This operation directs the process of 'ThreatAssesment',
*        'WeaponsAssignment', and 'WeaponsLaunching'.  Once a weapon has been
*        launched, its control is passed to a dedicated thread.
*
*    Recover - Recovers from a node failure by restarting 'TrackHandler' and
*        'WeaponsControl' threads.  Currently only recovers from the failure
*        of a single node.  A separate recover operation is invoked when I
*        node dies.  NOTE: If the node handling the recovery dies before it
*        has finished, the rest of the recovery will NOT be completed and
*        work will be lost.
*
* HISTORY
* -------
* 12-Dec-87  David Maynard (dpm) at Carnegie-Mellon University
*       Modified deadline constructs to meet new execution times.
* 20-Oct-87  Duane Northcutt (jdn) at Carnegie-Mellon University
*       Modified to use the new communications interface.
* 17-Aug-87  Russell Kegley (rbk) at General Dynamics
*       Added initial deadlines.
* 23-Jun-87  Betsy Zimmerman (betsy) at General Dynamics
*       Created.
*
\***********************************************************************/
```

```
/***************\
*
* INCLUDE FILES
*
\***************/

#include    "../h/c3demo.h"
#include    "../h/defs.h"
#include    "../h/trackdb.h"
#include    "../h/distrsvcs.h"
#include    "../h/wpns.h"


/********************\
*
* PERSISTENT STORAGE
*
\********************/

#undef DEBUG
#undef RBKBUG

WELLKNOWN CAPA SemaphoreManager SemaphoreManager;
WELLKNOWN CAPA ThreadManager    ThreadManager;
WELLKNOWN CAPA TempServices     TempServices;
WELLKNOWN CAPA CommIntfType     Comm;
WELLKNOWN CAPA DistrSvcMgrType  DSM;

CAPA TrackDBMgrType     TrackDB;
CAPA WeaponsMgrType     WeaponsMgr;
CAPA TrkHandlerType     TrkHandler;
CAPA ThreatAssessType   ThrtAssess;
CAPA TrkIdentifierType  TrkID;
CAPA LocalSvcMgrType    LSM;

/* Thread CAPA's */
CAPA Launch_AI_Thread;
CAPA Launch_GM_Thread;
CAPA Launch_SAM_Thread;

node_id_t       LogicalNodeID;

/* TrackQ Definitions */
#define TRACKQ_SIZE     100

CAPA Semaphore  TrackQSem;
CAPA Semaphore  TrackLockSem;

id_type trackq[TRACKQ_SIZE];

int     trackq_head,
        trackq_tail;
```

```
/**********************\
*
* OPERATION DEFINITIONS
*
\**********************/

OPERATION Init(IN CAPA TrkHandlerType     Handler,
               IN CAPA TrackDBMgrType     TDB,
               IN CAPA WeaponsMgrType     Weapons,
               IN CAPA ThreatAssessType   ThreatA,
               IN CAPA TrkIdentifierType  TrackID,
               IN CAPA LocalSvcMgrType    LocSM)
/*
* Initializes the TrackHandler.  Currently just accepts the CAPAs that
*  are needed for processing.
*/
{
    status_msg_t        stat;
    u_int               size;
    CAPA                DispatchThread;
    extern u_long       OCC_OO_TrkHandlerType_Dispatch;

    /* save CAPAs */
    TrkHandler = Handler;
    TrackDB = TDB;
    WeaponsMgr = Weapons;
    ThrtAssess = ThreatA;
    TrkID = TrackID;
    LSM = LocSM;

    /* initialize the track queue data structures */
    trackq_tail = 0;
    trackq_head = (TRACKQ_SIZE - 1);

    /* create the producer/consumer semaphore for the track queue */
    /*** NOTE:  Do not turn tracking on for this semaphore ***/
    INVOKE SemaphoreManager.AllocSem(IN (u_long)0,
                                     IN (boolean)FALSE, IN (boolean)FALSE,
                                     OUT (CAPA Semaphore)TrackQSem)
    ON FAILURE {
        sprintf(stat.buffer, "Handler.Init: sem create err R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        RETURN FAILURE;
    }
```

```
    /* create semaphore for locking new track queue */
    INVOKE SemaphoreManager.AllocSem(IN (u_long)1,
                                     IN (boolean)FALSE, IN (boolean)FALSE,
                                     OUT (CAPA Semaphore)TrackLockSem)
    ON FAILURE {
        sprintf(stat.buffer, "Handler.Init: locksem create err R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        RETURN FAILURE;
    }

    /* start the dispatch thread */
    INVOKE ThreadManager.Create(IN (CAPA)TrkHandler,
                                IN (u_long)OCC_OO_TrkHandlerType_Dispatch,
                                OUT (CAPA)DispatchThread)
    ON FAILURE {
        sprintf(stat.buffer, "Handler.Init: create thread err R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        RETURN FAILURE;
    }

    INVOKE TempServices.MakeReady(IN (CAPA)DispatchThread)
    ON FAILURE {
        sprintf(stat.buffer, "Handler.Init: make ready err R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        RETURN FAILURE;
    }
};
```

```
OPERATION ProcessTrack(IN id_type trackid)
/*
* Handles the processing of a new track that has entered the system.  New
*  tracks are identified, and if , handed off to the threat handler.
* Since we don't want to lose this activity if a node fails, this activity
*  is registerd with the 'TrackDBMgr'.
*
* Invoked by: PlotCorrelator.CorrelateFrame
*/
{
    int         abort_count;
    u_int       size;
    boolean     abort_flag;
    status_msg_t        stat;
    track_type          track_report;
    trackentry_type     tracktype;

    abort_count = 0;

begindl:
    abort_flag = FALSE;
    DEADLINE(6000, 300, 20, 0, 0, 8000, 0) {

        INVOKE LSM.GetNodeID(OUT LogicalNodeID)
        ON FAILURE {
            ERROR_HALT();
        }

        INVOKE TrackDB.GetTrackInfo(IN (id_type)trackid,
                                    OUT (track_type)track_report)
        ON FAILURE {
            ERROR_HALT();
        }

        INVOKE TrkID.IDTrack(IN (id_type)trackid,
                             IN (track_type)track_report,
                             OUT (trackentry_type)tracktype)
        ON FAILURE {
            ERROR_HALT();
        }

        INVOKE TrackDB.UpdateTrackType(IN (id_type)trackid,
                                       IN (trackentry_type)tracktype)
        ON FAILURE {
            ERROR_HALT();
        }
    }
    ON ABORT {
        abort_flag = TRUE;
        abort_count = (abort_count + 1);
    }
```

```
    if (abort_flag) {
        sprintf(stat.buffer, "Handler: ProcessTrack ABORT - FIXME %d",
                abort_count);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        /* handle this somehow! */
        if (abort_count < 4) goto begindl;
    }

    if (Hostile(tracktype)) {
        /* if the track is , process the threat */
        INVOKE TrkHandler.ProcessThreat(IN (id_type)trackid)
        ON FAILURE {
        }
    }
};
```

```
OPERATION ProcessThreat(IN id_type trackid)
/*
* Determines the correct reaction to a threat that for some reason needs to
*  be reconsidered.  This may happen when the threat is first identified, or
*  when a previous intercept attempt has failed.  This operation directs the
*  process of 'ThreatAssessment', 'WeaponsAssignment', and 'WeaponsLaunching'.
* Once a weapon has been launched, its control is passed to a dedicated
*  thread.
*/
{
    int          i,
                 size,
                 abort_count,    /* indicates how many times DL aborted */
                 threat_value;   /* perceived threat of track */
    short        site_index;     /* source of resource */
    boolean      unpaired,       /* TRUE while threat is unpaired */
                 pairable,       /* TRUE while there are options available for
                                  *  pairing the threat */
                 abort_flag;     /* indicates if DL aborted on this try */
    int_options_t favored,       /* weapons actually allocated */
                  assigned,      /* weapons actually allocated */
                  int_options;   /* intercept options to choose from */
    pairing_info_t pairings;     /* structure containing actual pairings */
    id_type      inter_track;    /* trackid of interceptor */
    track_type   inter_report,   /* dummy report entry for creating
                                  *  interceptor track */
                 track_report;   /* dummy report entry for getting
                                  *   track position */
    pos_type     track_pos,      /* coordinates of  track */
                 site_coord;     /* coordinates of site */
    status_msg_t stat;           /* buffer for status messages from Alpha */

    /* for now the decision of what type of resource to allocate *
     *  will be alternating between missiles and AI's */
    statictoggle = 0;
    abort_count = 0;

beginдl:
    abort_flag = FALSE;
    DEADLINE(6000, 900, 40, 0, 0, 8000, 0) {
        INVOKE TrackDB.GetTrackInfo(IN (id_type)trackid,
                                    OUT (track_type)track_report);
        track_pos.x = track_report.pos[LASTH(track_report.currpos)].x;
        track_pos.y = track_report.pos[LASTH(track_report.currpos)].y;
        track_pos.z = track_report.pos[LASTH(track_report.currpos)].z;

        INVOKE WeaponsMgr.InquirePairings(IN (id_type)trackid,
                                          OUT (pairing_info_t)pairings);

        pairable = TRUE;
```

```
        /* later on, when we want to get real sophisticated, we may
         * want to check the pairings that are left to see if they
         * are adequite. */

        unpaired = ((pairings.num_paired == 0) ? TRUE : FALSE);

#ifdef DEBUG
        sprintf(stat.buffer, "pairable=%d, unpaired=%d", pairable, unpaired);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
#endif DEBUG

        while (pairable && unpaired) {
            /* assess the threat potential of the track */
            INVOKE ThrtAssess.AssessThreat(IN (id_type)trackid,
                                           IN (track_type)track_report,
                                           OUT (int)threat_value);

            /* get a list of intercept options available to us */
#ifdef DEBUG
            sprintf(stat.buffer, "PROC Threat: Ready to GetInterceptOpt");
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
#endif DEBUG

            INVOKE WeaponsMgr.GetInterceptOptions(IN (id_type)trackid,
                                                  IN (pos_type)track_pos,
                                                  OUT (int_options_t)int_options);

#ifdef DEBUG
            sprintf(stat.buffer, "TrkHnd: Type %hd, #%hd, Type %hd, #%hd",
                    int_options.ai.intercepts[0].resource_type,
                    int_options.ai.num_intercepts,
                    int_options.gm.intercepts[0].resource_type,
                    int_options.gm.num_intercepts);
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
#endif DEBUG

            /* decide which interceptors are the best choices, i.e.,
             *  simulate c3 operations here. For right now just set
             *  'n_intercept' to 1, indicating we'll settle for the
             *  first choice returned from 'WeaponManager' */
            if ((int_options.ai.num_intercepts > 0) ||
                (int_options.gm.num_intercepts > 0)) {
```

```
#ifdef DEBUG
                sprintf(stat.buffer, "PROC Threat: Allocating Weapons");
                size = (STAT_MSG_HDR + strlen(stat.buffer));
                UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                              IN (u_int)size,
                                              IN (status_msg_t)stat);
#endif DEBUG

                /* allocate the weapon type */
                /* assess the response.  if we got what we asked for,
                 *  launch the requested weapons at the target track;
                 *  otherwise just release the allocated weapons and
                 *  reassess threat, options for interception, etc. */
                if ((toggle++) % 2) {
                    /* chose an interceptor (manned) */
                    int_options.ai.num_intercepts = 1;
                    int_options.ai.intercepts[0].resource_count = 1;
                    int_options.gm.num_intercepts = 0;
                    int_options.gm.intercepts[0].resource_count = 0;
                    INVOKE WeaponsMgr.AllocateWeapons(IN (int_options_t)
                                                      int_options,
                                                      OUT (int_options_t)
                                                      assigned);

                    favored.ai.num_intercepts = 1;
                    favored.ai.intercepts[0] = int_options.ai.intercepts[0];
                    favored.gm.num_intercepts = 0;
                }
                else {
                    int_options.ai.num_intercepts = 0;
                    int_options.ai.intercepts[0].resource_count = 0;
                    int_options.gm.num_intercepts = 1;
                    int_options.gm.intercepts[0].resource_count = 1;
                    INVOKE WeaponsMgr.AllocateWeapons(IN (int_options_t)
                                                      int_options,
                                                      OUT (int_options_t)
                                                      assigned);
                    favored.gm.num_intercepts = 1;
                    favored.gm.intercepts[0] = int_options.gm.intercepts[0];
                    favored.ai.num_intercepts = 0;
                }

                if (SameWeapons(&favored, &assigned)) {
                    for (i = 0; (i < assigned.ai.num_intercepts); i++) {
                        /* launch the assigned weapon at the target *
                         *  first get a specific controller object */
                        site_index = favored.ai.intercepts[i].site_index;
```

```
                    /* NOTE: the ID of the friendly is yet to be
                     *  determined, so NIL is passed in */

                    UNCHECKED INVOKE DSM.Request(IN (activity_t)LAUNCH_AI,
                                                 IN (short)site_index,
                                                 IN (id_type)NIL,
                                                 IN (id_type)trackid,
                                                 IN (short)INITIAL_LAUNCH,
                                                 OUT (CAPA)Launch_AI_Thread)
                    ON FAILURE {
                        sprintf(stat.buffer, "LAUNCH bad %d", STATUS);
                        size = (STAT_MSG_HDR + strlen(stat.buffer));
                        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STAT,
                                                      IN (u_int)size,
                                                      IN (status_msg_t)stat);
                    }
                }

                for (i = 0; (i < assigned.gm.num_intercepts); i++) {
                    site_index = favored.gm.intercepts[i].site_index;

                    /* NOTE: the ID of the friendly is yet to be
                     *  determined, so NIL is passed in */
                    UNCHECKED INVOKE DSM.Request(IN (activity_t)LAUNCH_GM,
                                                 IN (short)site_index,
                                                 IN (id_type)NIL,
                                                 IN (id_type)trackid,
                                                 IN (short)INITIAL_LAUNCH,
                                                 OUT (CAPA)Launch_GM_Thread)
                    ON FAILURE {
#ifdef DEBUG
                        sprintf(stat.buffer, "LAUNCH GM bad %d", STATUS);
                        size = (STAT_MSG_HDR + strlen(stat.buffer));
                        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STAT,
                                                      IN (u_int)size,
                                                      IN (status_msg_t)stat);
#endif DEBUG
                    }
                }
                unpaired = FALSE;
            }
            else {
                /* didn't get the weapon we wanted, so release the weapons
                 *  just allocated and try again */
                INVOKE WeaponsMgr.DeallocateWeapons(IN (int_options_t)assigned);
#ifdef DEBUG
                sprintf(stat.buffer, "Nothing launched, not same weapons");
                size = (STAT_MSG_HDR + strlen(stat.buffer));
```

```
                    UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                                  IN (u_int)size,
                                                  IN (status_msg_t)stat);
#endif DEBUG
                    unpaired = TRUE;
                }
            }
            else {
                /* no intercept option available */
                pairable = FALSE;
            }
        }

        if (unpaired == FALSE) {
            /* COMMIT SUICIDE - DO NOT RETURN */
            /* NOTE: check on what clean up needs to be done here */
        }
        /* NOTE: we now know there is no interceptor paired with the track */

        if (pairable == FALSE) {
            /* assuming infinite resources we should not see this case now.
             * later on probably alert operator who may be able to take
             *  action to reassign 'int_options' or make other options
             *  available. */
             /*** TBD ***/
        }
    }
    ON ABORT {
        abort_flag = TRUE;
        abort_count = (abort_count + 1);
    }

    if (abort_flag) {
        sprintf(stat.buffer, "Handler: ProcessThreat ABORT - FIXME %d.",
                abort_count);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        /* handle this somehow! */
        if (abort_count < 4) {
            goto begindl;
        }
    }
};
```

```
OPERATION Dispatch()
/*
* This is the entry point for a captive thread which actually dispatches
*  the DSM requests for new tracks.
*/
{
    status_msg_t        stat;
    u_int               size;
    boolean             abort_flag = FALSE;
    id_type             next_track;
    CAPA                track_thread;

#ifdef DEBUG
    sprintf(stat.buffer, "Handler.Dispatch Ready");
    size = (STAT_MSG_HDR + strlen(stat.buffer));
    UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                  IN (u_int)size,
                                  IN (status_msg_t)stat);
#endif DEBUG

    while (TRUE) {
        /* wait for something to be put into queue */
        INVOKE TrackQSem.PSem()
        ON FAILURE {
            sprintf(stat.buffer, "Handler.Dispatch Queue Fail R=%d S=%d",
                    RESULT, STATUS);
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
        }

        DEADLINE(6000, 800, 100, 0, 0, 9000, 10) {
            /* lock the new track queue */
            INVOKE TrackLockSem.PSem()
            ON FAILURE {
                sprintf(stat.buffer, "Handler.EnterTrack: Lock fail R=%d S=%d",
                        RESULT, STATUS);
                size = (STAT_MSG_HDR + strlen(stat.buffer));
                UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                              IN (u_int)size,
                                              IN (status_msg_t)stat);
            }

            /* remove track from head of queue */
            trackq_head = ((trackq_head + 1) % TRACKQ_SIZE);
            next_track = trackq[trackq_head];

            /* unlock the new track queue */
            INVOKE TrackLockSem.VSem()
            ON FAILURE {
                sprintf(stat.buffer, "Handler.EnterTrack: Unlock fail R=%d S=%d",
                        RESULT, STATUS);
```

```
                size = (STAT_MSG_HDR + strlen(stat.buffer));
                UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                              IN (u_int)size,
                                              IN (status_msg_t)stat);
            }

            /* ask DSM for a process track activity */
            UNCHECKED INVOKE DSM.Request(IN (activity_t)PROCESS_TRACK,
                                         IN (id_type)next_track,
                                         OUT (CAPA)track_thread)
            ON FAILURE {
                sprintf(stat.buffer, "Handler.Dispatch: DSM fail %d %d",
                        RESULT, STATUS);
                size = (STAT_MSG_HDR + strlen(stat.buffer));
                UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                              IN (u_int)size,
                                              IN (status_msg_t)stat);
            }

#ifdef RBKBUG
            sprintf(stat.buffer, "TrkHandler.Dispatch: dispatched %d",
                    next_track);
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
#endif RBKBUG
        }
        ON ABORT {
            abort_flag = TRUE;
        }

        if (abort_flag) {
            abort_flag = FALSE;
            sprintf(stat.buffer, "Handler.Dispatch: DL ABORT - FIXME");
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
        }
    }
};
```

```
OPERATION EnterTrack(IN id_type trackid)
/*
* Enters a new track in the queue of new tracks.  The dispatch thread will
*  actually ask the DSM for a 'ProcessTrack' activity on the least-loaded
*  node.
*/
{
    status_msg_t        stat;
    u_int               size;

#ifdef RBKBUG
    sprintf(stat.buffer, "TrkHandler.EnterTrack: entering track %d",
            trackid);
    size = (STAT_MSG_HDR + strlen(stat.buffer));
    UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                  IN (u_int)size,
                                  IN (status_msg_t)stat);
#endif RBKBUG

    /* lock the new track queue */
    INVOKE TrackLockSem.PSem()
    ON FAILURE {
        sprintf(stat.buffer, "Handler.EnterTrack: Lock fail R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
    }

    /* check for queue full */
    if (trackq_head == trackq_tail) {
        /* queue is now full (or EXTREMELY close since we do synchronize),
         *  signal error to let people know there is a problem.  This
         *  is actually an early warning, but we don't want this queue
         *  getting anywhere near full. */
        sprintf(stat.buffer, "Handler.EnterTrack: Queue FULL - FIXME");
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
        INVOKE TrackLockSem.VSem()
        ON FAILURE {
            sprintf(stat.buffer, "Handler.EnterTrack: Unlock fail R=%d S=%d",
                    RESULT, STATUS);
            size = (STAT_MSG_HDR + strlen(stat.buffer));
            UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                          IN (u_int)size,
                                          IN (status_msg_t)stat);
        }
        RETURN FAILURE;
    }
```

```
    /* enter new track at tail of queue */
    trackq[trackq_tail] = trackid;

    /* advance tail pointer */
    trackq_tail = ((trackq_tail + 1) % TRACKQ_SIZE);

    /* unlock the new track queue */
    INVOKE TrackLockSem.VSem()
    ON FAILURE {
        sprintf(stat.buffer, "Handler.EnterTrack: Unlock fail R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
    }

    /* indicate that there is another item in the queue */
    INVOKE TrackQSem.VSem()
    ON FAILURE {
        sprintf(stat.buffer, "Handler.EnterTrack: Queue Fail R=%d S=%d",
                RESULT, STATUS);
        size = (STAT_MSG_HDR + strlen(stat.buffer));
        UNCHECKED INVOKE Comm.SendMsg(IN (msg_t)MSG_ALPHA_STATUS,
                                      IN (u_int)size,
                                      IN (status_msg_t)stat);
    }
};
```

```
/******************\
*
* LOCAL PROCEDURES
*
\******************/

int     SameWeapons(requested, assigned)
    int_options_t       *requested;
    int_options_t       *assigned;
/*
* Return TRUE iff requested weapons match assigned weapons.
* For now a crude check for the counts being equal is used.
* Later on, each element of the requested list will have to be
*  checked against the assigned list.
*/
{
    if ((requested->ai.num_intercepts == assigned->ai.num_intercepts) &&
         (requested->gm.num_intercepts == assigned->gm.num_intercepts)) {
        return (TRUE);
    }
    else {
        return (FALSE);
    }
};


int     Hostile(trktype)
    trackentry_type     trktype;
/*
* Return a TRUE if the given track is a  type
*/
{
    if ((trktype == HOST_BOMBER) || (trktype == HOST_MISSILE) ||
         (trktype == HOST_JAMMER)) {
        return(TRUE);
    }
    else {
        return(FALSE);
    }
};

}       /* end of object */
```