The Alpha Operating System: Requirements and Rationale

J. Duane Northcutt

TR #88011

Archons Project Technical Report

January 21, 1988

The Alpha Operating System: Requirements and Rationale

J. Duane Northcutt

January 21, 1988

Archons Project Technical Report #88011

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

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 a new kind of operating system, which is unique in two highly significant ways—first, it is *decentralized,* providing reliable resource management transparently across physically dispersed nodes, so that distributed applications programming can be done 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) which are characterized by predominately aperiodic activities, having critical time constraints (such as deadlines) associated with them. Alpha is extremely adaptable so as to be easily optimized for a wide range of problem-specific functionality, performance, and cost.

Alpha is oriented towards systems having on the order of 10 to 100 nodes, which are physically dispersed on the order of 1 to 1000 meters. The Alpha operating system is for the most demanding kind of situation: mission-oriented systems where all nodes are contributing to the same application, not simply for the network case of individual users at each node doing unrelated computations. The focus of this research is on having nodes be logically integrated together, rather than autonomous. Alpha provides this logical integration by executing on the bare hardware and managing resources in the same sense as a uniprocessor operating system does, not by being just a "UNIX-style" user process and providing standard application interfaces and protocols for simple inter-node resource sharing, as is done in conventional, computer network-style distributed operating systems. Resources must often be managed by Alpha across node boundaries in the best interests of the whole application, not just on the usual per-node basis. This necessitates that Alpha also accept responsibility for handling asynchronous concurrency and reliability issues which arise in distributed systems, instead of passing them all up to the users for recurring, lower performance solutions. Alpha provides facilities which are necessary and sufficient to maintain consistency of data and correctness of operation at both the system and application levels, despite concurrent execution and node or communication path failures. This is accomplished through the use of techniques similar to those normally found, far above the operating system, in distributed database systems—e.g., nested atomic transactions, replication. With Alpha, the system's constituent nodes collectively form a single computer, not a computer network; thus, distributed application software can be written as though it were for a conventional uniprocessor—without the applications programmer even knowing about, much less having to manage, distributed resources.

The term "real-time" is usually intended to mean "deterministic behavior" and "faster is better," particularly in the area of interrupt handling and context swaps. Real-time control in this sense applies only to computer systems which simply do low-level sensor/actuator sampled-data loop applications, and are traditionally designed to have rigidly periodic behavior. But supervisory real-time control is far more difficult because it encompasses not just such static periodicity but also predominantly dynamic and aperiodic activities which nonetheless have critical time constraints, such as deadlines. These constraints are part of the correctness criteria of the computation, and failure to meet them is a threat to the systems's mission and to survival of property and human life.

A novel approach is taken in Alpha whereby the time constraints of each of an application's constituent activities are expressed in terms of the value to the system of completing each activity as a function of its completion time (deadlines are a simple special case of this, that is, a

unit value step function). In addition, computational activities have relative importances which are also time-dependent. These time-value functions and importances are dynamic and are used by the system in the time-driven management of system resources. The conventional and seemingly simpler notions of "priority" in real-time systems are poor approximations to this approach, and extensive experience has consistently demonstrated that priorities introduce massive and uncontrollable complexity into all but the most trivial real-time systems. Alpha employs this new real-time management technique to resolve *all* contention for resources such as processor cycles, communication access, secondary storage, and synchronizers (e.g., semaphores and locks). Time constraints and importance are among the attributes propagated with computations which cross node boundaries so that resource management can be global. The ubiquitous client/server model is unsuitable in this respect since it does not maintain such essential correspondences between the service and client on whose behalf that service is being provided.

Furthermore, Alpha exhibits a fundamental philosophy which is contrary to that of operating systems for other application environments. Instead of optimizing performance of the normal cases at the expense of infrequent ones, it does the opposite. It is in the exception cases such as emergencies (e.g., being in danger due to failure or attack) when a real-time operating system must be depended upon to perform best, even if the system's routine performance must be suboptimal to ensure that. This is one of the principal reasons why the expression "real-time UNIX" is inevitably an oxymoron.

Supervisory real-time control applications are very complex, and are not (perhaps cannot be) well understood; in addition, the environment and technology are always in a state of flux. Thus, the functional and performance requirements for their computers evolve continuously throughout the entire life cycle of the system (which can be decades). Alpha accommodates this situation through a variety of techniques, many of which are quite innovative. Its design is kernelized and strictly adheres to the principle of policy/mechanism separation. Specific operating system policies are carefully excluded from its kernel level mechanisms so that a wide range of different service facilities, and indeed entire decentralized operating systems, can be effectively constructed using Alpha's kernel, in accordance with application needs. For example, Alpha's kernel provides atomicity, serializability, and permanence of operations as orthogonal mechanisms. Conventional atomic transaction facilities bundle all three properties together, with correspondingly high overhead, as the only choice of policy regardless of need and affordability. But the client layers of Alpha's kernel can base their policies on other combinations of these mechanisms. For example, there are many instances in real-time systems when problem-specific consistency constraints yield correct results more efficiently than serializability would, or when permanence is not worth its cost. This same philosophy is followed in scheduling, communications, and all other types of system resource management.

Computers embedded in real-time control systems usually must produce the highest possible performance from the allowable hardware size, weight, and power, including memory space for the operating system. A general-purpose computer system can easily be an order of magnitude lower in performance than a special-purpose one for a particular application. Thus, to achieve the balance of performance and flexibility needed for cost-effectiveness in a multiplicity of changing real-time applications, Alpha is general-purpose but unusually malleable so as to exploit all the problem-specific static and dynamic information available from the application. In addition, application functionality can readily be migrated downward into the operating system, and even into its kernel, for increased performance when necessary.

Alpha's internal implementation is organized so that its subsystems (such as scheduling, communications, secondary storage, etc.) can all execute truly concurrently within each node. It is intended that these separate hardware points of control within Alpha be a mixture of dynamically assigned general-purpose processors (i.e., each node in the decentralized computer can be a multiprocessor) and algorithmically specialized hardware accelerators (co-processors and other forms of augmentation). Alpha extends to its client applications the same opportunities for taking advantage of multiple special-purpose as well as general-purpose processors at each node.

Alpha presents a programming model which is object-oriented, in the sense of supporting basic programming units which adhere to the definition of abstract data types. This imposes a structure and discipline conducive to modular software at both the operating system and application levels, as well as improving fault isolation. The active entity, or unit of logical computation, is known as a *thread,* whose execution progresses through objects via operation invocation, without regard for address spaces or node boundaries; distribution and reliability are the responsibility of Alpha instead of the user. This network uniformity and transparency greatly aid the creation and modification of distributed applications.

Alpha is the first systems effort of the Archons Project, and the prototype was created at Carnegie-Mellon University directly on multiprocessor nodes constructed with Sun workstation hardware [Northcutt 88b]. It has been demonstrated with a real-time control application written by its first industrial user, General Dynamics [Maynard 88]. An enhanced, second-generation, commercial-quality version being produced at Kendall Square Research is portable but initially targeted at one of their multiprocessor products. Both versions of Alpha are sponsored by the USAF Rome Air Development Center and are in the public domain for U.S. Government use.

1 Introduction

The overall goals of the Archons project involve the exploration of both architectural and operating system issues related to real-time command and control. The Archons project began following a major hardware construction effort (i.e., the HXDP [Jensen 78a] system constructed at Honeywell's Systems and Research Center), is now in the midst of an operating system construction effort, and plans are being made for the next phase of hardware construction.

The research described in this document is based on a significant amount of theoretical study and substantial experience with a variety of actual real-time command and control problems and systems. The Alpha operating system is being developed as the first implementation effort of the Archons project. This empirical effort is an important phase in the project's overall "think-do" strategy—i.e., a research approach based on a repeating cycle of conceptualizing and theorizing, designing and implementing, then measuring and evaluating.

The Archons work is focused on pushing in new directions to achieve greater benefits than is available by obtaining the next increment from some aging technology. Alpha is not merely an academic exercise, but a validation of the project's conceptual efforts. Real-time command and control is an area that is well understood by the members of the Alpha team (which represents over 70 years of collective experience in the real-time area), as is the need for research to advance the state of the art in the area. Alpha is directed more towards meeting the needs of this problem area than simply generating academic publications.

Real-time command and control applications demand a new kind of operating system which directly addresses their special requirements in the areas of: meeting user-specified real-time processing constraints, supporting complex distributed applications, survivability under hostile attack, and adaptability to evolving needs across (frequently decade-long) system lifetimes. These characteristics all demand that the operating systems needed for real-time command and control differ drastically, not just in degree but in kind, from conventional ones, calling for dramatically innovative concepts and techniques. Very little technology has, or will, come from the mainstream academic and industrial operating system research and development community, which is primarily focused on completely different types of applications (e.g., personal productivity workstations, throughput-oriented numerically intensive computing, and time-independent transaction processing).

The real-time command and control application area has vital requirements which strongly differentiate it from the context in which almost all academic research and commercial product development on operating systems takes place. First, most of the activities in the system (and therefore in the computer) have stringent time constraints which are a matter of correctness rather than convenience, and responsiveness to these constraints, not throughput, is the primary metric of performance. Second, the safety of human life and property depends on the survivability of the system, despite its being subjected to warfare. Third, even though the system consists of physically dispersed elements, they are all dedicated to cooperatively performing a single mission, instead of being simply a network of communicating but otherwise independent and unrelated individual entities. Finally, the system must be adaptable and evolvable to accommodate significant changes in the mission, the environment, and the technology base across a period of decades, often while remaining in non-stop service.

Another crucial, but often overlooked, factor in achieving the most cost-effective real-time operating system is that the performance of the system must be optimized for the high-stress exception cases, such as emergencies caused by hostile attack or faults, even if the normal case performance must be suboptimal to accomplish that. This is diametrically opposed to the prevailing approach in non-real-time operating system design and implementation, where performance is optimized for the most common case at the expense of less frequent cases.

The programming model that was developed in this effort stems from an examination of the requirements of the problem area, the in-depth study of several key techniques for solving the fundamental problems, and the application of sound engineering trade-offs in combining the individual features into a coherent system. This is different from other operating system efforts in a number of ways. The design of Alpha was derived top-down, from high-level requirements and not bottom-up, from a set of implementation details. Most significantly, 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 splice the newly developed features into an existing system.

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. There is a unique combination of requirements implied by this application domain, and existing systems either do not attempt to meet these requirements at all, or do not do so in a comprehensive, well-integrated fashion. The Alpha programming model provides features that meet each of the main requirements of the defined applications domain.

This is a technical overview of the Alpha operating system, with emphasis on the features of systems research that are frequently overlooked, such as the context within which Alpha is being created, and the assumptions upon which this work is based. In this document, the Alpha operating system's intended application domain is defined, and the special requirements of the chosen problem area are described. This report goes on to motivate the project's work by exploring the ways in which existing operating systems do not adequately address the needs of this area, and the technical approach taken in Alpha is presented in brief. Finally, some of the rationale for the design of the Alpha system is given and specific features of the system are related back to the defined requirements.

2 Application Domain

This section describes the application domain from which the Alpha operating system's requirements were derived. The intent of this description is to explicitly define the assumptions upon which this work has been done, in order to provide a basis upon which to state the rationale for the Alpha operating system and to provide a context in which the work can be judged.

2.1 Distributed Systems

Because of the widely differing definitions of "distributed systems" that abound in the literature, it is important that the definition used in this document be made explicitly clear. For the purposes of this research, a *decentralized computer* is considered to be a machine that consists of a multiplicity of physically dispersed processing nodes, integrated into a single computer through a native, global *decentralized operating system* [Jensen 78a] (as shown in Figure 1). A decentralized operating system manages the system's collective, disjoint physical resources in a unified fashion, for the common good of the whole system. This permits a distributed system's disjoint resources to be applied directly to a single application.

A logically singular, yet physically dispersed computer is highly appropriate in many contexts (e.g., real-time command and control). In particular, decentralized computers provide independent failure modes for reliability, multiple execution units for concurrency, and a flexible interconnection structure for extensibility.

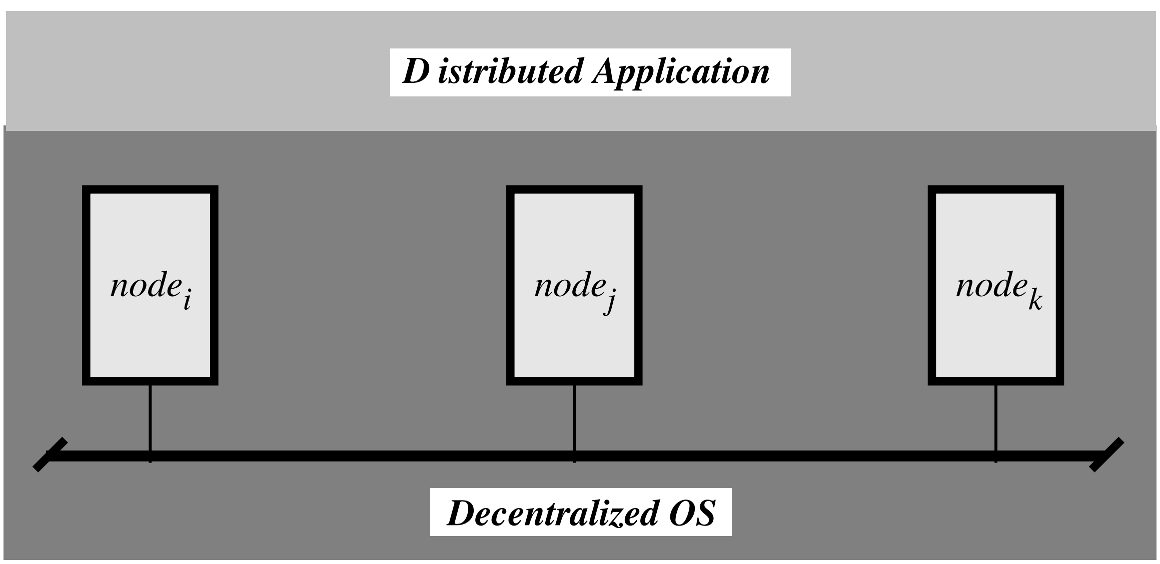

Figure 1: Example Decentralized System

It is often necessary or desirable that an application be distributed across multiple nodes in a decentralized computer system. A distributed application is far more integrated than individual applications which share resources or even interact. A distributed application is conceptually a singular entity created with a coherent, consistent perspective and set of objectives, not separate entities which each have their own independent objectives without concern for, or even awareness of, others. The constituent computations of a distributed application may extend across physical node boundaries (for a variety of reasons), rather than each being confined to its own local address spaces and the resources on one node.

A decentralized operating system consists of replicated copies that constitute the native operating system on all the system's processing nodes. However, the system's resources are managed in a global fashion. This is done either by direct coordination among the various instances of the kernel, or locally by each kernel instance as consequences of the higher-level resource management decisions.

These system characteristics are quite different from those of more traditional multiprocessors, computer networks, and other systems with similar hardware structures. For example, in computer networks (such as shown in Figure 2) the resources at a particular processing node are managed by the operating system local to that node, and the collection of autonomous, local operating systems interact in limited ways (e.g., to support such application purposes as file sharing, mail, and remote login) [Lampson 81]. Furthermore, the emphasis in the work described here is on multi-computer systems which do not have shared primary memory, separating it from operating systems built on multiprocessor hardware [Jones 79, Ousterhout 80, Wulf 81].

It should be noted that the hardware structure of a decentralized computer system, as defined here, need not be different from that of typical local area networks (i.e., a collection of processing elements with private memory and local peripheral devices, interconnected by an interconnection network). It is usually the operating system software alone that separates decentralized computer systems from local area networks.

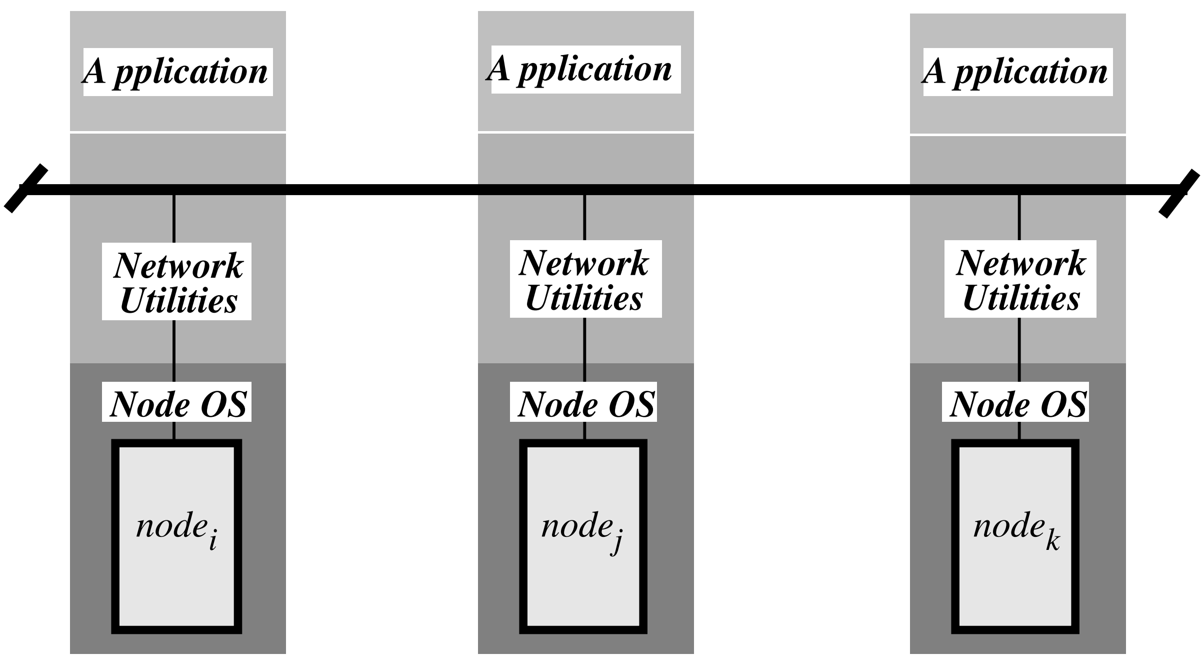

Figure 2: Example Distributed System

The class of distributed real-time command and control systems of interest in this research involves on the order of 10 to 100 processing nodes, physically dispersed across a distance on the order of 10 to 1000 meters.

2.2 Real-Time Command and Control

According to the definition used in this research, real-time computer systems control one or more physical processes whose states are sensed and altered by the computer system. The state of these processes changes independently as a result of the external environment, which is not completely under the control of the computer system. The principal distinction of real-time computing systems is that the physics of the application imposes time constraints (e.g., deadlines) on some of the computations. These constraints are not simply performance metrics, or an issue of programmer convenience, but are part of the correctness criteria for those computations.

The emphasis in real-time systems today is on a very specific subset of the general real-time application domain, and it is this area that is most typically assumed when the term “real-time” is used. The special case that the greatest amount of effort has been directed towards is the area of the lowest level sampled-data monitoring and control of physical processes—i.e., dataflow/pipelined signal processing or sensor/actuator feedback control. Furthermore, because these applications are inevitably considered to be rigidly periodic and deterministic, this popular view of real-time systems is not even an accurate perception (much less an optimal treatment) of the intended case.

There is a great deal more to real-time systems than is encompassed by the typical low-level view, however the larger real-time picture is based on the general notion that arbitrary computations may have constraints on their times of execution. In the most general real-time case, the application’s time constraints may vary dynamically, the time constraints may not be predicted ahead of time, and not all of the computations that make up an application have time constraints associated with them at all times.

The class of real-time system considered in this research is known as *supervisory-level command and control*. Supervisory control is a middle-level function in the real-time application hierarchy—above the sampled data loop functions and below the human interface/management functions (see Figures 3 and 4). The area of real-time supervisory control is representative of the characteristics of the larger real-time application domain. Supervisory control systems do little direct polling of sensors and manipulation of actuators, nor do they provide extensive man/machine interfaces; rather, they interact with subsystems which provide those functions. In addition to having activities with critical time constraints, the behavior of real-time command and control systems is predominantly dynamic (i.e., aperiodic, asynchronous, and stochastic). Supervisory real-time command and control systems are found in plant (e.g., factory or refinery) automation, vehicle (e.g., airborne, aerospace, or shipboard) control, and surveillance (e.g., air-traffic control) systems.

Some tasks in real-time command and control systems are periodic and are bound to process activity rates, but most are aperiodic with stochastic parameters and are associated with external stimuli as well as the interactions of the system with low-level control subsystems [Boebert 78]. The real-time response requirements of a supervisory control system are closer to the millisecond than either the microsecond or second ranges.

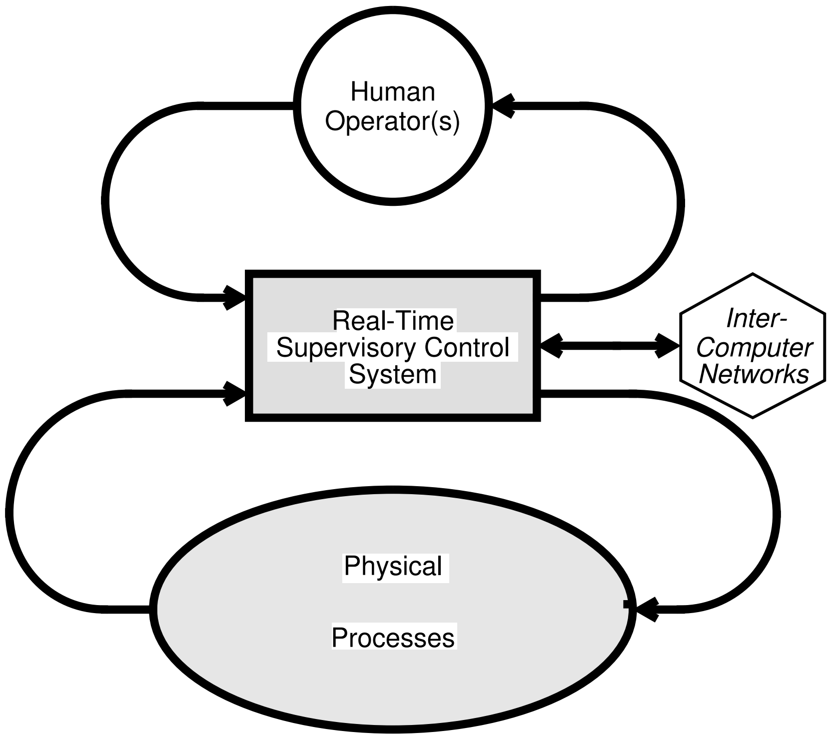

Figure 3: Real-Time Supervisory Control Context

Many command and control applications (and their I/O subsystems) are physically dispersed, and for this reason map well onto distributed computer systems. Furthermore, the distributed nature of these command and control systems suggest that they be implemented as distributed application programs. The concurrently executing computations within these distributed applications exhibit more cooperative behavior than those among other distributed applications. This is especially true of real-time command and control applications, where the concept of independent users does not exist as it does in typical timesharing and computer network systems [Thomas 78]. Rather, the whole system is dedicated to performing a particular mission that can be thought of as having a single user, comprised of the physical processes being controlled.

Level	Function	Operating System	Processor
4	Management Information System	VM (IBM)	3083 (IBM)
3	Plant Management System	VMS (DEC)	8800 (DEC)
2	Supervisory Control System	Alpha	Sun-3 (SMI)
1	Machine Control System	VRTX (H&R)	11/23 (DEC)
0	Sensors/Actuators	*none*	*various*

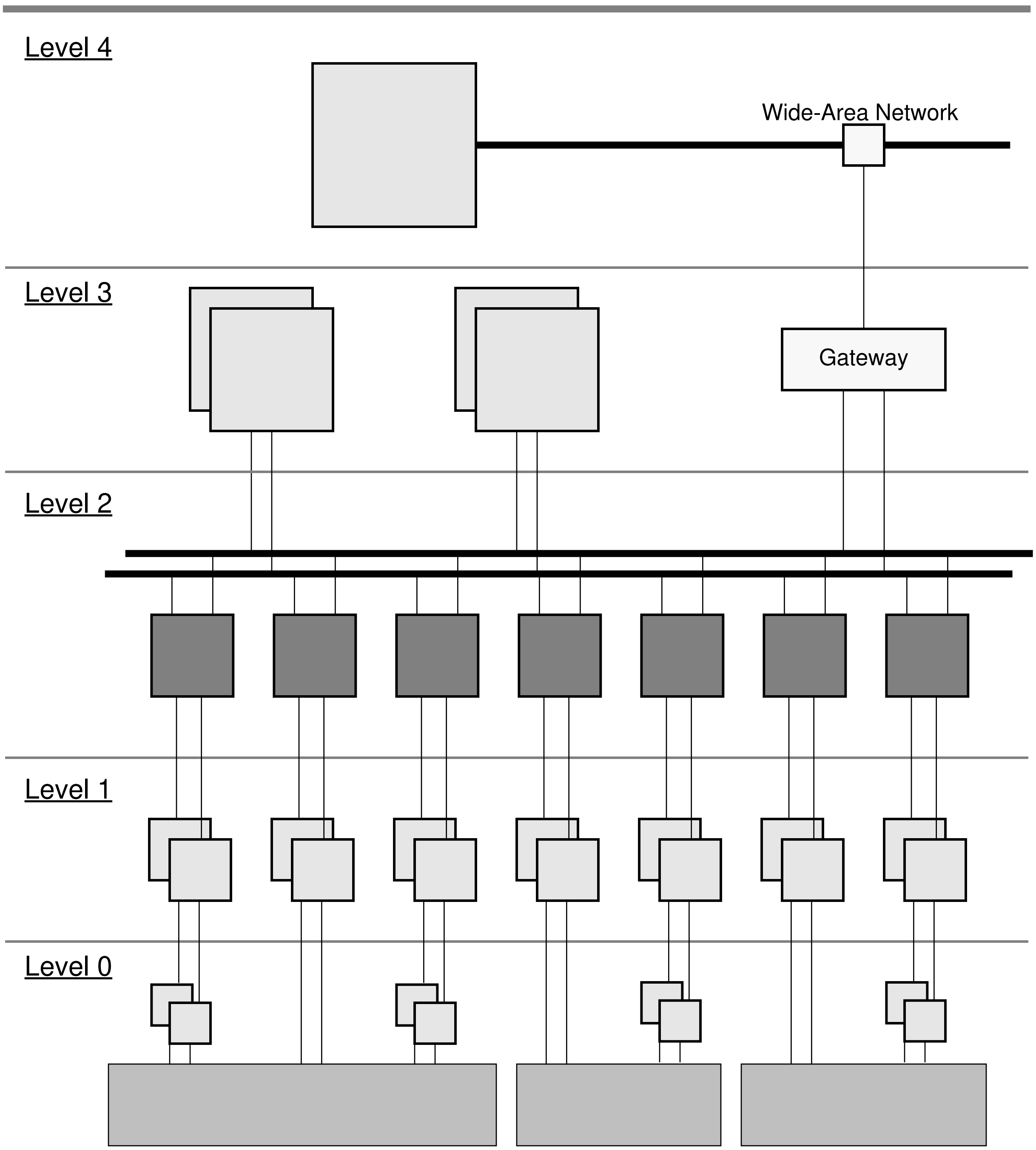

Figure 4: Example Real-Time Systems Hierarchy

3 Special Requirements

A number of significant requirements accompany the distributed real-time command and control application domain. Some of these requirements are unique to this context, and others, while generally applicable to a wide range of systems, are especially important here. It is these application requirements that form the basis from which the Alpha operating system's programming abstractions and supporting facilities were derived. The requirements of this application domain can be grouped into four categories—i.e., features associated with the timely execution of programs, issues related to the physical distribution of the system, matters concerning the robustness of the system and application code, and the demands for adaptability at both the system and client levels.

3.1 Timeliness

In a real-time system, computations must be performed in accordance with the time constraints imposed by the physical processes being controlled. Furthermore, because of the dynamic nature of the external processes being controlled by a real-time system, some amount of the application's state information is time-dependent and its value degrades with time. For these reasons, the timeliness of the computational activities performed by a real-time system is an integral part of the definition of correct system behavior. In a real-time system it is not sufficient to ensure only that data generated as a result of a computation is correct and consistent, it must also be made available in a timely fashion for the result to be considered correct.

An operating system that is to support real-time applications must provide mechanisms which take these time-related issues into account and assist application programs in meeting their time constraints [Jensen 76b, Wirth 77]. There is a range of degrees to which a system can support the needs of real-time applications. A system could make it difficult to meet real-time demands (e.g., by trying to enforce a "fairness" policy in resolving contention for resources), it could provide mechanisms that help in the construction of real-time applications, or it could be structured so as to attempt to make real-time guarantees for its applications. Clearly, the term "real-time" does not define a simple dichotomy among operating systems, but rather there is a range of metrics with which any given operating system can be judged as supporting the needs of real-time applications. The more that a system does to enhance the ability of client programmers to create applications that meet their timeliness constraints, the greater claim the system has to being "real-time."

There are a number of standard requirements associated with all real-time systems; the least recognized, but most important of these is the notion of time-driven resource management. All of these requirements are discussed in the following subsections, along with some of the commonly held misconceptions about the general requirements for real-time operating systems.

3.1.1 Standard Requirements

Irrespective of their technical merit, there are some widely accepted requirements for real-time operating systems. Among these requirements are a number of features that are common features of modern operating systems that, because of advances in computer technology, have only recently become practical for use in real-time operating systems (or are just being discovered by practitioners in the area of real-time systems—see [Glass 80]).

Real-time operating systems have been receiving a great deal of attention in recent times, however this attention has largely resulted in the rediscovery of features found in traditional, full-featured operating systems (e.g., dynamic multiprocessing, interprocess communications, address space protection, and high-level synchronization mechanisms). This is similar to how much of the work in microprocessors has resulted in the rediscovery of features found in traditional, full-featured mainframes (e.g., I/O channel processors and memory management).

In addition to those requirements that are common to modern operating systems in general, there are those which are primarily performance enhancements—e.g., fast context swaps, low interrupt latency, and high-bandwidth I/O. Other requirements affect only the implementation and are not directly visible to the client—e.g., allowing portions of the system to be paged, allowing preemption while executing within the kernel, providing for the streaming of data to/from contiguous regions on disk, and bounding execution times for system services. Some of these commonly held requirements have to do primarily with the system's implementation, but are visible to the client—e.g., the ability to pre-allocate system resources, and the ability to *lock* code/data into memory (i.e., prohibit the system's virtual memory subsystem from removing a region of primary memory). Yet other of these standard requirements are generally useful features that are visible to the client—e.g., the ability to access I/O devices directly from user space, the ability for clients to define (and load) custom device drivers, the ability to force writes to disk, and the ability to be asynchronously notified of specific I/O, exception, and application-specific events.

These standard requirements are largely taken for granted in the Alpha operating system research effort. These requirements are recognized as being important to the development of practical real-time systems, and the significant effort required to meet them is not underestimated. However, the focus of this research is on the other requirements defined in this chapter, because of the relatively large amount of experience and relatively small degree of complexity that the more common requirements exhibit with respect to the other requirements (e.g., distributed, reliable, time-driven resource management).

3.1.2 Time-Driven Resource Management

Fundamental to the definition of all operating systems is the management of system resources, and similarly the major factor in providing support for real-time applications has to do with how an operating system manages its resources. To support the needs of real-time applications, an operating system must manage resources in such a fashion as to attempt to ensure, to the greatest extent possible, that the timeliness constraints of its application programs are met. In addition to managing system resources in an efficient and consistent fashion (as all operating systems should), if the system manages its resources based directly on the actual time constraints of the application, this is known as *time-driven resource management*. According to this definition, the primary issue in the creation of real-time systems is the amount and degree of time-driven resource management that they perform.

In the extreme case, whenever a real-time operating system makes a decision as to how an instance of contention for some resource is to be resolved†, it considers the impact of its decision on the timeliness of the computations that may be affected. Operating systems that resolve instances of system resource contention based on a metric other than timeliness (e.g., fairness or throughput), do not work to meet the application's time constraints and are not as qualified

†*Note that the manner in which resources are managed is not significant if there is no contention for the resources—i.e., if there are enough resources to meet all of the demands.*

to be termed real-time systems.

An implication of time-driven resource management is that the operating system must be given (or be able to infer) the application's time constraints, and must be able to determine the impact of its resource management decisions on the application. In general, an operating system can only perform crude, brute-force types of resource management without (either explicit or inferred) guidance from the application. An adaptive, real-time operating system must receive (run-time or compile-time) inputs from its clients in order to resolve contention for system-managed resources in the desired fashion. Clearly, the more information that the operating system has, the better it can do in managing resources so the application's time constraints can be met. Similarly, the closer that an operating system's abstractions are to the programmer's abstractions, the less transformation must be performed (either by the programmer or the system) to map the application's timeliness requirements into the form needed by the system to manage its resources, and the easier it is for the operating system to determine the effects of its management decisions on the application.

3.1.3 Common Misconceptions

In addition to these requirements, there are some commonly held misconceptions concerning the requirements for real-time operating systems. Some of these beliefs are derived from valid requirements of specialized subsets of the general real-time application domain. However, most of these mistaken beliefs stem from primitive attempts at dealing with the difficult problems of real-time systems by attempting to cast the problem into a form that is more intellectually manageable, or that might be amenable to the application of some existing analytical tool. The most serious of these myths are that priority-based schedulers adequately support the needs of real-time applications, that real-time applications are predominately periodic in nature, and that poor system resource utilization can be used as an effective substitute for time-driven resource management.

3.1.3.1 Priority-Based Processor Scheduling

A common misconception about real-time systems is that all applications' timeliness requirements can be met with a simple priority scheduler. Firstly, it should be noted that scheduling is not the only resource management activity that an operating system performs. Because application software makes requests for resources primarily when a software unit has been dispatched, and because there is contention for processor cycles whenever more than one schedulable software entity in a system is ready to run, the management of processor cycles (i.e., scheduling) is an important and frequent resource management activity in any operating system.

The use of a priority scheduler does not go very far towards supporting the needs of real-time applications. In the form they are most often found, priority schedulers exhibit a number of problems. For example, when dealing with periodic activities, the notion of priority is derived (more or less directly) from application time constraints—e.g., static rate-monotonic priorities, and dynamic deadline priorities. However, the time constraints of aperiodic activities tend to be mapped in a much less methodological fashion into a narrow range of small integers intended to signify relative priority. This is because aperiodic activities are less amenable to analytical treatment, and integer priorities are thought to be more computationally tractable than actual time constraints.

There are numerous deficiencies with the typical, *ad hoc,* uses of priorities. Included in the list of problems is the fact that much valuable information is lost in crude transformations from the time domain and into priorities, such as when a single metric is used to express two orthogonal characteristics of a computational activity—e.g., the timeliness requirements of a computation and the importance of its execution to the application. Furthermore, the range and resolution of the priority scale is often too limited to accommodate the number and gradation of characteristics of application activities. Fixed priorities also represent a premature binding of the dynamic information associated with computations that denies the system the opportunity to adapt to the many important changes possible at execution time—e.g., stochastic variations in load and resource contention, and faults, errors, and failures. The assignment of fixed priorities to computational units in a system requires a great deal of trial-and-error manipulation to achieve the desired effects. Furthermore, in some cases there does not exist a fixed priority assignment which allows all the activities to meet their time constraints, even though the system's resources might be under-utilized (see Figure 5).

Priorities which are dynamic overcome many of the adaptability and flexibility restrictions of static priorities, but even dynamic priorities (such as deadlines) suffer from serious disadvantages. Among the problems with dynamic priorities are: not begin able to represent both the urgency and importance of computations, not being able to express both the *hard* and *soft* varieties time constraints, not having the power and expressiveness to depict a variety of different types of time constraints, and not being able to deal gracefully with overload conditions.

The real-time environment addressed by Alpha demands a type of scheduler that supports a dynamic expression of the timeliness constraints of each computation. Such a scheduler must also be able to determine how to allocate processor cycles to each of the application's schedulable software entities in such a way as to meet the application's time constraints whenever possible, and follow the system's overload handling policy when all of the application's time constraints cannot be met.

3.1.3.2 Deterministic Systems

Another myth is that real-time systems must behave in a highly predictable fashion in order for the system's performance "guarantees" to be met [Stankovik 88]. While many systems rely on the assumption that real-time applications are strongly periodic, the rigidly cyclic style in which typical (low-level) real-time systems are designed is rarely due to any inherent property of real-time applications. Instead, periodicity is only a particular technique which is believed implify the management of actual time constraints. .

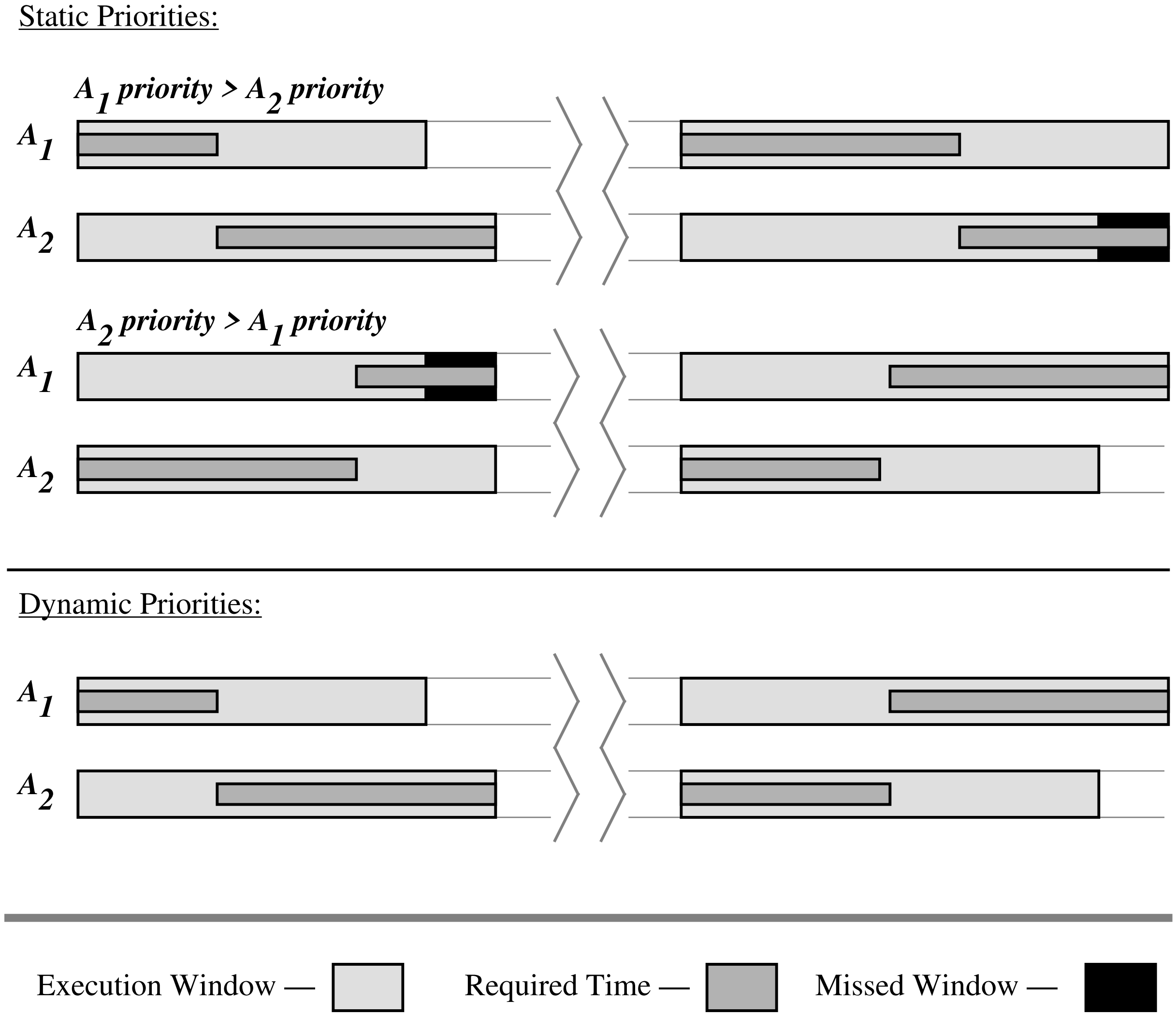

Figure 5: Static versus Dynamic Priorities

This is despite the fact that many real-time applications include devices (and thus computations) which appear to exhibit periodic behaviors, such as: sensors which generate data on a physically cyclic basis; actuators which require recurring positional updates; and displays which need to be refreshed regularly**.** The classical periodic approach to expressing and handling such characteristics is to define a fixed iteration period for the cyclic, recurring, or regular activity, schedule all future iterations of that activity collectively in advance, and mutually synchronize (if possible) all periodic activities (e.g., into rate-groups) [Leinbaugh 80].

General real-time command and control systems and their applications cannot be constrained to behave in a highly deterministic fashion. Only certain, very stylized, computations (e.g., signal processing) exhibit such rigidly predictable behavior, and the underlying (possibly distributed) system cannot be constrained to behave in such an idealized fashion. Some periodic approaches are based on highly deterministic environmental assumptions—e.g., no exceptions occur during the system's execution, there are always sufficient resources to meet the application's needs, and activities always take the same amount of time to complete. Operating systems that are based on such simplistic and unrealistic assumptions do not meet the needs of the greater space of real-time control problems. For larger and more demanding distributed real-time command and control systems, it is less reasonable to make such assumptions because the behavior of these systems is necessarily much more complex, dynamic, and stochastic, nor is it possible to constrain the system to behave in the desired fashion.

It is quite difficult to construct a system that is adaptive and yet able to meet demanding timeliness constraints, because adaptive distributed algorithms are difficult to create and tend to be costly in terms of performance. Nonetheless, such techniques must be developed to deal with the true needs of general real-time control applications.

3.1.3.3 Excess Assets

Another real-time myth is that an excess of system resources is sufficient to meet the demands of real-time applications. The belief that excess resources solves real-time application needs stems from the fact that if there is no contention for (or low-utilization of) system resources, then the resource management strategy employed by the operating system is not significant. It is true that the brute force approach of providing a sufficiently high degree of excess hardware assets, alone, would be capable (in principle) of meeting the computational needs of virtually any real-time applications. However, the complete suite of requirements for actual real-time systems cannot be met with such a simplistic approach. It is the demands of other system attributes, such as cost, size, weight, and power, that require system resource management schemes which do not depend on low resource utilization to meet application-level timeliness constraints. Also, practical experience shows that applications expand to fit the size of their systems, and that there are rarely enough resources to meet the basic needs of the application, much less the degree of excess assets needed to meet timeliness constraints in this highly non-cost-effective manner.

3.2 Distribution

One of the major functions of decentralized operating systems and network operating systems is to manage the inter-node communication resources for the clients. But in a distributed system, it is highly desirable that physical dispersal of the underlying hardware be made transparent at a low level in the system (i.e., the kernel). In this manner, both the system and the application programmers benefit from features such as physical-location-transparency, similar to how programmers benefit from a system-provided process abstraction. However, there are a number of cases (e.g., work assignment, redundancy management, specialized function location, or diagnostics) in which it is appropriate to provide clients of the kernel with information concerning physical location.

In a distributed system, system and application software is normally based on *thin-wire* [Metcalf 72], as opposed to shared memory, interconnection techniques. This suggests that performance, availability, and robustness would suffer if system mechanisms were to make use of centralized structures. Furthermore, the physical dispersal of the system hardware introduces variable and unknown communication delays that exacerbate the already difficult task of attempting to ensure deterministic behavior of the system. The kernel mechanisms for decentralized systems must therefore be adaptive and deal explicitly with the effects of physical dispersal—i.e., inaccurate and incomplete information. Although it is below the kernel level and deals with a small number of simple, static resources, the ARPANET routing algorithm [McQuillan 80] provides an example of this type of behavior.

Integrating physically dispersed hardware into a logically unified system at the operating system level means that the clients of a decentralized operating system should not be required to be aware of, much less manage, the physical dispersion of hardware and software, nor the many complex consequences thereof. These issues should not be allowed to distract the clients from performing their application tasks. A decentralized operating system supports distributed applications without requiring users to perform the necessary distributed resource management. Such an operating system provides the client programmers with a coherent distributed programming model. It also provides the management of physically dispersed resources necessary to transparently integrate the system's processing nodes into an actual (not just apparent) single system. All of these functions should be performed by a decentralized operating system, without depending on any form of physically, or even logically, centralized control.

3.3 Robustness

In the context of this effort, robustness means the extent to which the application continues to function fully and correctly (including meeting the system's timeliness requirements) in the face of failures (which is meant to include *faults*, *errors*, and *failures* as defined in [Avizienis 78]).

3.3.1 Implications of Distributed Real-Time Control

The nature of the physical processes being controlled in real-time command and control systems is usually such that the safety consequences (in terms of personal or property damage) of not managing them properly can be quite severe. Thus, the robustness of a real-time control system is of utmost importance, often being more significant than the cost or performance of the system.

Furthermore, it is often the case that the controlling real-time system is embedded and cannot be maintained easily. Should a component fail, it can be quite difficult (if not impossible) to gain access to the system to perform the necessary repairs, and frequently repairs must be made without interrupting the normal functioning of the system. The *mission* times—i.e., the period of time when the system must continue to provide correct service, uninterrupted by either maintenance or repair activities—of real-time systems can range from hours to years, and there is no single adequate approach to dealing with the issues of robustness posed by each of these cases.

A decentralized operating system for the real-time control environment has special robustness requirements which are more elaborate than those for conventional network and distributed operating systems. For example, the function of providing physical location transparency of software modules is not complete without handling the automatic detection of, and supporting recovery from, node and interconnection failures. Furthermore, a decentralized operating system is itself a set of distributed programs and special-purpose distributed (both replicated and partitioned) databases. This fact has major implications on the construction of the system, and calls for the use within (not just above) the operating system of robustness techniques inspired by traditional distributed databases, such as atomic transactions and the automatic management of replicated data.

To support the overall robustness goals of a distributed real-time command and control application, the system software must itself meet a certain level of robustness. In addition to this, the system must provide mechanisms that allow suitably reliable applications to be constructed. The operating system should not dictate a specific kind and degree of robustness, but rather it should allow its clients to choose what is desired for each individual set of circumstances, at a cost that is appropriate. Such a flexible, mechanism-based approach is conceptually more difficult to design, but supports increased system efficiency, in that the operating system's clients are free to make the appropriate cost/functionality tradeoffs on a case-by-case basis within their application programs.

3.3.2 Major Robustness Concepts

The distributed real-time command and control application domain calls for a set of operating system mechanisms that support the following robustness concepts: correctness of data and actions, availability of data and services, graceful degradation of function, and fault containment. While these concepts are useful in almost any system, they are critical in this application domain.

3.3.2.1 Correctness of Data and Actions

The correctness of the actions performed by an application is a function of time, sequencing, and completeness—i.e., the correctness of a set of actions is defined by the amount of time it takes each action to execute, the order in which actions execute, and whether, at the end of the set of actions, they were all successfully executed. The correctness of data is defined by its self- and mutual-consistency as well as its timeliness of visibility. It must be possible for the programmer to ensure that actions which manipulate data leave it in a consistent state, despite any failures that may occur in the course of the data manipulation.

To provide for the correctness of data and actions, an operating system must support some form of atomic transaction facility [Lampson 80, Moss 85]. Such a facility ensures the *atomicity*, *permanence*, and *serializability* attributes of atomic transactions are made to apply to computations that execute within the defined boundaries of atomic transactions.

3.3.2.2 Availability of Services and Data

The availability of services and data is defined to be the extent to which each service (and the data it encompasses) remains available to clients across system failures. Typically, a service that is statically and uniquely bound to a particular hardware functional unit becomes unavailable should that hardware unit fail. To increase the availability of services, failures that may result in service disruption must be eliminated, or the means must be provided for resuming the service elsewhere when a failure occurs. The dynamic nature of the physical processes being controlled implies that a great deal of information in a distributed real-time control system degrades with time. This dictates that the availability of information is at least as important as, if not more important than, its consistency. Under some circumstances in real-time applications there is little value in maintaining the consistency of a database if the information is unavailable for use when needed. Similarly, when the correctness of the information in a database degrades over time, there may be little point in attempting to restore it to a meaningful consistent state following a period of unavailability.

For an operating system to provide for enhanced availability of services and data, some facility must be included to manage redundant copies of code and data—i.e., some type of replication scheme [Bayer 78] must be supported by the system.

3.3.2.3 Graceful Degradation of Function

Graceful degradation of function is defined to be the property of a system that permits it to continue providing the highest level of functionality possible as the demand for resources exceeds its currently available supply. In the context of this work, whenever contending requests for resources cannot all be met in an acceptable time, the contention should be resolved in favor of the functions that are most critical to the objectives of managing the physical processes being controlled. Information concerning the relative importances of individual application tasks can be provided only by the application programmer. To provide graceful degradation, the system could use this importance information to determine which tasks' needs will not be met, in order to sustain the highest and most useful level of functionality possible under a given overload condition.

In addition to providing a scheme for load-shedding under overload conditions, a real-time operating system should be based on hardware with separate, independent failure modes (e.g., a distributed system) in order to provide support for graceful degradation of functionality. In addition, the operating system must provide mechanisms for node failure detection and recovery (e.g., location-transparent communications and migration/reconfiguration mechanisms).

3.3.2.4 Fault Containment

Fault containment is defined to be a property that inhibits the propagation of errors among system components. If a failure occurs (or is induced) in a system or application component, the operating system mechanisms should limit (or assist its clients in limiting) the extent to which the failed component can adversely affect the behavior of others. For example, the operating system should not permit a failed software component to arbitrarily modify the state of other components, either intentionally or by accident. (This point of view is in direct opposition to the current trend toward "light-weight processes.") Furthermore, for this property to hold, mechanisms must ensure that a failed software component cannot consume resources in an unconstrained manner and thereby interfere with other, presumably good, software components.

While some degree of fault containment can be provided at compile-time, in many cases the only effective way of enforcing the containment of faults is by way of system-provided hardware mechanisms (e.g., a memory management unit). In order to achieve the benefits of fault containment, the price of enforcement must be paid. While it is clear that most systems can be made to run faster if they do not provide defensive protection measures, it is not as clear that the robustness increase is not worth the price paid in terms of performance (especially in systems where robustness is a critically important attribute).

3.3.3 Optimizing for Exception Cases

An additional robustness characteristic of real-time systems is that the correct and timely execution of a real-time command and control application is typically more important under exception conditions than in normal cases. Also, the nature of exceptions (such as hardware failures due to physical damage) in these systems is such that they tend to be clustered in both time and space—as opposed to being randomly (e.g., normally) distributed, as is commonly presumed in other contexts. These characteristics have a significant impact on the nature of the fault tolerance and recovery techniques used in systems, and are contrary to the premises underlying almost all non-real-time computing system approaches (e.g., RISC-style operating system philosophies and the "end-to-end" argument) [DRC 86]. The designer should make an effort to perform reasonable engineering tradeoffs in the system design to ensure that the proper emphasis is placed on the exception conditions and not have the system be optimized exclusively for the expected, non-exceptional cases.

Figure 6 provides an simplified illustration of the types of tradeoffs typically made in non-real-time systems (6a), and the objective that real-time systems should aspire to (6b). It should be noted that a higher degree of steady-state overhead may be tolerated by real-time systems in order to reduce the magnitude and variance of performance resulting from exceptions.

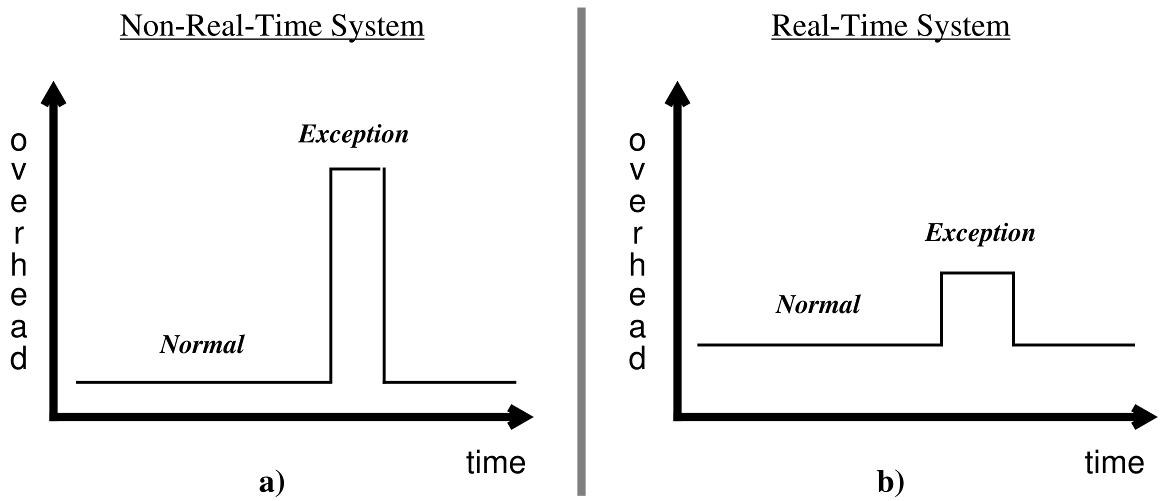

Figure 6: Optimizing for Exception Cases

3.4 Adaptability

In real-time command and control systems, modifications, technology upgrades, testing, maintenance, and other life-cycle items invariably make up the most significant portions of a system's cost [Savizky 85]. Typically, the software-related costs predominate because the requirements are poorly understood at system design time and continue to evolve throughout, not just the design and implementation phase of the system, but even during the system's lifetime (which may be a decade or more) [Boehm 81, Lehman 85]. The principle reasons for this are that: the application is extremely complex and is not (perhaps cannot be) well understood; the application environment (e.g., the product or threat) varies according to economics, geopolitics, doctrine, etc.; computer technology changes rapidly, making some desirable system attributes possible and others more cost effective, but also making older technology unaffordable or inaccessible; and the computing system is frequently viewed as the ultimate recourse for accommodating system changes (because modifications are always just "a simple matter of programming").

This implies the need for system software to provide a programming model that supports such desirable software engineering attributes as adaptability and maintainability. Because these attributes are more frequently associated with languages than operating systems, an operating system for real-time command and control might facilitate adaptability by taking into account the run-time packages of languages that address these attributes of adaptability.

In the realm of real-time command and control control, system sizes range from quite small (e.g., an individual vehicle), to very large (e.g., an entire plant). Furthermore, in a decentralized system, processing nodes may be added and removed (either statically off-line, or dynamically due to the run-time failure and recovery of hardware). The operating system must itself be able to function effectively across a wide range of application and system sizes to take advantage of the opportunities for extensibility offered by the inherently modular hardware architecture of a distributed system, and to provide the robustness available from reconfiguration.

The need for the properties of modularity and extensibility are not unique to decentralized computer systems. Centralized systems may provide the necessary level of performance for a given application and may incorporate system software that is quite modular and extensible; however, centralized uniprocessor and multiprocessor hardware pose quite severe limitations in these respects. Decentralized systems offer a wider range of cost/performance choices than is usually available from a family of centralized processors or shared-memory multiprocessors. Overall, the physical properties of decentralized systems offer the potential for greater robustness, extensibility and adaptability, both within the system as well as in the applications [Franta 81].

Modularity and extensibility (even more so than robustness) are considerably more difficult to quantify and measure than other system attributes, such as performance. This may account for the fact that concern for performance (usually in the form of throughput) often dominates other considerations, fostering the misconception that these other system attributes are of lesser importance. This is unfortunate since system performance increases are derived automatically from advances in semiconductor technology, while increases in system robustness, adaptability, and extensibility result almost exclusively from thoughtful effort by system designers.

4 Current Practice

Operating systems in existence today (even those claimed to be "real-time") do not adequately address the requirements posed by the type of distributed real-time command and control applications described here. While some systems deal with certain aspects of the overall problem, it is safe to say that none meets even a significant number (much less all) of them. In the following sections, current operating system practice is examined with respect to each of the previously defined major requirements areas for real-time command and control applications.

4.1 Timeliness

It is not reasonable to characterize operating systems simply as being real-time or not, but rather there is a spectrum, into which all systems fit, that defines the degree to which the system can meet the demands of various real-time applications. The aspects of operating systems that increase their suitability for real-time applications are not manifest merely in the inclusion or exclusion of specific functions, nor does the inclusion or exclusion of any functionality, in itself, make an operating system unsuitable for real-time use.

Today there is range of meanings for the term "real-time operating system," as evidenced by the widely disparate types of systems that are referred to as being real-time. The applicability of this term to a particular piece of system software can best be judged by examining two fundamental aspects of the system—namely, the extent to which it is an operating system (i.e., the kind and degree of resource management that it performed), and the extent to which it is real-time (i.e., the kind and degree of support for meeting users' time constraints that is provided).

Today's real-time "operating systems" vary widely in the kind and degree of resources they manage, programming models they support, and the degree to which undesirable artifacts of the underlying hardware are obscured. For example, the types of system software that is sometimes referred to as "real-time operating systems" range from simple rate-monotonic dispatchers (e.g., for avionics [GD 80]), to minimum functionality embedded-system executives (e.g., VRTX [Ready 86]), to communication-based non-real-time kernels (e.g., V [Cheriton 84]), to full-functionality non-real-time operating systems (e.g., UNIX).

The "real-time" operating systems of today also vary in the extent to which they support the needs of real-time programming. For example, things claimed to be real-time operating systems range from non-real-time systems (e.g., UNIX and V), to non-real-time systems with "real-time" extensions (e.g., RTU [Henize 86]), to non-real-time systems with extensions plus kernel enhancements such as preemptability (e.g., MACH [Accetta 86]), to priority-based "faster-is-better" systems (e.g., VRTX), to static, periodic-based dispatching systems (e.g., SubACS [Wallis 84]), to highly general and adaptive real-time operating systems (e.g., Alpha [Northcutt 87]).

4.1.1 Minimalist Systems

To an overwhelmingly large extent, the system software that is referred to as "real-time operating systems" today is aimed at solving problems at the low end of the spectrum of real-time applications. These tend to be minimal-functionality systems, based on the assumption of periodic application behavior, they frequently use static priorities to resolve contention for resources, and make use of low resource utilization in order to obtain hollow performance "guarantees."

This type of real-time "operating system" is merely vestigial compared with the normal meaning of the term, and are often more accurately termed "executives." Some such minimal systems (like those found in current avionics computer systems) are little more than rate-group dispatchers, and most provide only a library of rudimentary resource management routines. For the most part, they provide minimal functionality—preferring instead to pass the time, space, and intellectual complexity burdens of system resource management on to the application programmer.

These small executives strive to avoid doing anything that would make it difficult for applications to meet their time constraints, and try to provide service to the clients in a predictable fashion (primarily with respect to time); they incorporate little more than priority interrupt handling and context swapping facilities in an attempt to facilitate real-time responsiveness. The higher levels of functionality are typically left to the users in the belief that they can implement exactly what they need as a part of the application software, thus reducing execution time and memory consumption (however, this strategy frequently backfires).

Minimalist real-time operating systems arose from a historical context when processor cycles and memory space consumed a far greater proportion of the system's cost/size/weight/power than they do now. Such simple system software approaches may be adequate when the application software is correspondingly elementary, but the evolutionary trend (even in the area of avionics) is toward rapidly increasing system functionality and complexity. The effects of this trend can result in recurring, non-standard, lower performance attempts by users to perform system (as well as application) resource management—an especially expensive mistake with distributed systems.

4.1.2 Priority-Based Scheduling

The conventional approach to dealing with timeliness in current real-time systems is based almost exclusively on one or more of the following functions: static, priority-based processor scheduling, priority interrupts (typically statically assigned); fast interrupt handling and context swaps; fixed-priority (e.g., rate-group) scheduling, performed based on *a priori,* static preallocation of resources; and extensive configuration analysis, simulation, testing, and tuning. Notably missing from this list is anything that assists in the time-driven management of resources. Of all of the techniques used to make a system be real-time, by far the most common one is priority scheduling.

Instead of dealing directly with the application's time constraints, current real-time computing systems attempt to map them into artifacts which they (usually implicitly) imagine to be simpler, but often there are wide-spread detrimental consequences whose cause is not recognized. The principle forms that these artifacts take are the use of static priorities and the dependence on strongly periodic applications behavior. Such systems provide only a priority mechanism for managing processor cycles; the difficult task of making the priority assignments is left to the client, and typically requires a great deal of tuning. It is also common to pre-define multiple sets of fixed priority assignments and overlay them to meet certain anticipated combinations of changes (or modes) in the system.

Attempting to devise appropriate priority assignments for an application's computations is often an extremely complex and time-consuming effort that is relegated to highly-experienced specialists, and is (almost necessarily) dealt with in an *ad hoc* fashion. Furthermore, when changes are made to components of an application constructed using static priorities, it is frequently the case that the entire process of priority assignment and testing must be repeated. In addition to these deficiencies, it has been shown that, in general, fixed priority assignments are incapable of meeting activity deadlines, even when the computing capacity is far greater than the needs of the activities [Liu 73].

Provided that certain (often rather unrealistic) assumptions are met, some static priority-based scheduling techniques can be used to meet the time constraints of some types of real-time applications. For example, the *rate-monotonic* technique is known to be an optimal static algorithm with respect to meeting periodic deadlines [Liu 73]. However, only trivial systems manage to achieve a successful balance between priority responsiveness and resource utilization.

The solution to many, but not all, of these deficiencies is to use a dynamic priority scheme. However, despite the fact that dynamic priority assignments can significantly out-perform static ones, such techniques are rarely used in actual real-time systems.

4.1.3 Low Utilization

Some real-time systems attempt to guarantee that all of an application's timeliness requirements are always met. While it would be ideal for a real-time operating system to provide such timeliness guarantees, current approaches to making them introduce significant programming constraints and usually require that the system conform to certain unrealistically over-simplified assumptions concerning its behavior (e.g., as in [Leinbaugh 80]). Most efforts in this area focus on providing high resource utilization for low-level (i.e., closed-loop, sampled data) control applications where tasks are deterministically periodic and have no value to the system if their deadlines cannot be met (i.e., tasks have hard deadlines). The appeal of such approaches is their analytical tractability, but they are not suitable for more general real-time command and control contexts, which are characterized by predominately aperiodic tasks and are less amenable to such rigid and stylized treatment. In general, it is currently no more practical to make absolute timeliness guarantees for general, unconstrained real-time systems than it is practical to prove the correctness of large, complicated programming systems. Recently, some of the vocal academic proponents of real-time guarantees have begun recognizing the futility of their position and now speak of "conditional guarantees" [Stankovik 88].

4.2 Distribution

Despite the fact that physically dispersed systems are becoming more wide-spread, virtually none of them attempts to be globally unified in the previously defined manner. Most distributed systems are simply networks, or at most federated (dedicated-mission) systems, functioning as loosely associated collections of autonomous computing nodes with the ability to interact with each other [Thomas 78, Lampson 81]. In typical distributed systems, each node manages its local resources independently, and the client's interface to the system is usually non-location-transparent and non-uniform.

Ideally, the programming of distributed applications should be focused on the intrinsic issues of logical structure and algorithms, but distributed programming does depend on distributed resource management—i.e., the management of physically dispersed resources so as to maintain application correctness, consistency, and performance in the context of such complexities as: asynchronous (real) concurrency; variable, unknown communications delays; and multiple independent failure modes. Distributed resource management in this sense should be the responsibility of the system so that the users can devote their efforts to the application programming. However, most conventional distributed operating systems manage system resources locally (i.e., per-node) and only a few resources (e.g., inter-node communications) are managed globally across the whole system. Therefore, almost all of the distributed resource management for distributed applications must be provided by the users, at substantial recurring development costs, and at similarly high, recurring, performance penalties.

In most distributed systems today, the programmer of distributed applications must explicitly be aware of, and deal with, the effects of physical distribution (e.g., the physical locations of software modules, the number of nodes in the system, and current state of the each node). This is because most conventional "distributed" operating systems are simply centralized, local operating systems with additional standardized interfaces and protocols intended for certain forms of resource sharing among separate applications executing in a network of autonomous nodes—e.g., shared file access or remote procedure calls to servers. Usually this sharing of resources is not uniform and transparent to the applications; the system's clients must access remote resources in a manner much different from how local resources are accessed. Furthermore, the interfaces and protocols which are added to standard operating systems to make them "distributed" are typically implemented as utilities at the higher layers of the operating system. While this design decision serves to minimize the impact of distribution on local operating systems, it provides support for distribution at a higher cost than "native" implementations.

Typical centralized operating systems are extended to be "distributed" by adding a remote file system, or remote procedure calls between client and server processes. The resulting system provides a programming model that is inadequate for distributed applications in that it does not address some of the most fundamental issues in distributed programming, leaving them instead to be dealt with at the application level. First, such distributed systems typically do not provide a strong correspondence between the application's viewpoint of a computation which spans multiple nodes and the system's viewpoint of separate computations on interconnected nodes. To the underlying system, the components that make up the logical computations in a distributed application program are disjoint entities; the system does not see any relationship among them and therefore must manage system resources without the benefit of knowing about the relationships that bind them into a common logical computational activity. In addition, it is incumbent upon the application programmer to transform the structure of his logical computations into components that are supported by the system. The programmer's logical view of a computation is the only thing that holds these independent components together—they all look the same to the system that is managing resources for them.

In addition, the most common type of distributed system does not provide the means for propagating a logical computation's attributes (e.g., real-time constraints and reliability needs) along with it, as it progresses through the various components it makes use of in the course of its execution. This means that the system does not maintain the continuity of the acquired attributes of a logical computational stream as it passes from one module to the next (regardless of whether it involves crossing node boundaries). This makes it impossible for the system to maintain the continuity of the information that is associated with a computation, as opposed to the various components that go into making up each computation in an application. Because the system does not maintain this continuity, it cannot manage resources effectively and there is only a limited degree to which an application can make up for this deficiency.

Most distributed systems in existence today tend to provide limited, stylized support for concurrency and synchronization of distributed computations. These systems tend not to provide support for either concurrency within a (multiprocessor) node, or concurrency among the nodes in a system. But rather, they provide the type of process management and interprocess communication facilities that were developed for uniprocessor or network systems. Additionally, the forms of synchronization dictated by these systems tend to limit the amount of concurrency that can be obtained within a programming module. This is typically done because the systems that these structures were developed for did not have any actual concurrency to exploit, and so restricting concurrency has the beneficial effect of simplifying things for the programmer. If the system restricts concurrency within a programming module to where only one point of control can be active in it at a time, the programmer need not be concerned about synchronization, and can write the usual form of non-reentrant, straight-line, serial code.

4.3 Robustness

A common technique for attempting to provide robustness in real-time control systems is through the extensive use of excess assets—i.e., the system is provided with far more resources than necessary to meet the application's (steady-state) computational demands in order to ensure that the system's objectives can continue to be met in the face of failures. These systems rely on a very low level of average system resource utilization in order to achieve their robustness goals.

Most extant systems that address the robustness requirements described in the previous sections tend to be special-purpose database systems that use low-concurrency, high-cost consistency maintenance schemes (e.g., atomic transactions), or equally costly redundancy management techniques (e.g., replication). The use of these techniques has not been directed towards the area of distributed real-time command and control, and so little effort has been expended in trying to develop robustness techniques that also meet the timeliness demands of real-time systems. Frequently (regardless of the specific techniques used) a system's robustness facilities impose substantial constraints on the system and application programmers, and furthermore, most of the systems that have been constructed to explore techniques for reliability emphasize only one aspect of system reliability. The reliability facilities in current systems tend to impose fixed costs for their use, irrespective of whether the client currently needs the system's robustness services. Furthermore, the reliability facilities currently being developed are not geared to meet the special needs of the real-time command and control area, and are therefore not well integrated with the other facilities required by this application domain.

Regardless of the particular mechanisms used, a common technique for ensuring that a real-time application exhibits the proper degree of robustness is exhaustive testing. Despite the fact that it is widely known that testing only detects the presence of bugs and does not ensure the absence of bugs, testing is used to obtain a greater level of confidence in systems (and configurations). As it is currently practiced, the exhaustive testing of real-time systems is a dead-end technology, that is already being pushed to its limits, and will not hold up to the demands of future systems. As real-time command and control systems have become more complex, it has become impossible to exhaustively test them. With larger numbers of processors carrying out larger numbers of tasks, the combinatorial explosion of paths does not allow simple exhaustive tests to be written and applied with any degree of confidence that a high degree of coverage can be obtained. As systems become more dynamic, it is no longer possible to anticipate, with a small number of load modules, all of the system configurations that may result from subsystem failures. The testing issue has long acted as a force to keep real-time systems small, simple, and deterministic. What is called for is the advancement of the state-of-the-art in testing (as well as design-for-testability) technology to cope with the advancing sophistication of the systems instead of using testability as an excuse for constraining the systems.

4.4 Adaptability

While certain existing systems have emphasized modularity and extensibility in their designs, they have not been systems of the type that is of interest in this research. Although some real-time control systems exhibit a significant degree of architectural modularity (via the use of processing nodes interconnected by high-performance communication networks [Jensen 78b]), and others provide adaptability at the programming language level (via a high-level language such as Ada), there are few examples that have explicitly attempted to be usefully modular in their system-level design. Likewise, there are operating systems that provide a high degree of modularity and extensibility at their interfaces and within their structure, however few of these have been distributed systems and fewer still have been for real-time command and control.

4.4.1 Dynamic System Behavior

Currently, most real-time systems attempt to make the behavior of the system and applications as static and deterministic as possible, and attempt to validate the correctness of the system through exhaustive testing [Quirk 85]. A common characteristic of this type of system is the high cost associated with the addition or modification of system functionality, and the inability to cope with unanticipated behavior [Parnas 77, Glass 80].

In real-time control environments the value of flexibility has not always been properly appreciated because the computing system is usually embedded. The pervasive philosophy is that a static system (that is analyzed, configured, tested, tuned, and validated *a priori* for as many contingencies as possible) meets all of the needs of real-time applications, is more dependable, and maximizes performance with minimum assets. However, factors now mitigate in favor of greater computer system flexibility than has historically been available in this environment. Many applications are becoming so sophisticated that the common static approach is clearly infeasible, and computer hardware technology advances have diminished the size, weight, and power per unit of performance, as well as the percentage of system costs represented by computer hardware assets.

4.4.2 Dynamic Time Constraints

The greatest impediment to adaptability in real-time systems results from their approach to managing time constraints—most take an approach that depends on periodic behavior of the system to make the problem more manageable, or amenable to analytical treatment. As outlined in the requirements section of this report, real-time applications themselves are not necessarily so strongly periodic. While there are many cyclic activities in real-time systems, the actual time constraints for many types of repetitive-sample activities can be expected to have a significant degree of variability (e.g., due to the mechanical tolerances of the physical environment). Even if the application does impose strictly periodic requirements, there are aspects of the application's implementation and the underlying system that introduce variance into the system. This periodic-based approach is not required by real-time applications, it does not adequately address all of the real-time application problems, and it restricts the flexibility of systems in a number of different ways.

In particular, the periodic approach does not deal with the variability that accompanies real systems. Many forms of variability are inherent in any (non-trivial) real-time system—e.g., stochastic fluctuations in load and resource contention, mechanical tolerances in sensors and actuators, and faults, errors, and failures of system components. A periodic-based approach attempts to, in an *a priori* fashion: anticipate certain variabilities and define away the majority of those that remain; prohibit architectural optimizations that introduce nondeterminism (such as caches, virtual memory, and direct memory access channels); design, and provide resources, for the “worst case”; dedicate and pre-allocate all system (and application) resources; test and tune the system until all cases tested “work”; and “guarantee” system behavior (i.e., response times). This results in rigid, brittle structures which must be invested with massive quantities of resources, and these assumptions must not be violated or else the system will shatter catastrophically under the pressure of variabilities (e.g., ancient bridges and cathedrals which were rigid and massive).

Furthermore, a periodic-based approach does not allow the run-time evolution of time constraints. In periodic-based systems, once the period of a repetitive activity has been *a priori* established, tested, tuned, and validated, it must remain fixed despite any run-time changes that may occur in the application environment. There are many cases where this is a serious limitation. For example, where the interval between successive repetitions may evolve dynamically throughout the course of a periodic activity’s execution—sampled-data monitoring/positioning can occur with increasing frequency as a robotic manipulator approaches an object, a pair of controlled vehicles approach each other, or a variable gets closer to its limit.

Additionally, the periodic-based approach to real-time system construction does not deal properly with aperiodic events. In many important real-time environments (such as real-time command and control) time-critical aperiodic activities are a major factor, and are the predominant form of time constraint. Conventional periodic-based real-time systems attempt to force-fit aperiodic activities into their periodic mold—e.g., by providing excess capacity reserved for aperiodic activities in a background frame, or by providing a periodic server which multiplexes aperiodic activities. But none of these schemes is capable of effectively scheduling activity mixes in which a significant percentage are aperiodic activities having time constraints (such as deadlines), or efficiently utilizing resources.

Both periodic and aperiodic activities fit naturally in an aperiodic-based approach, thus it is the more general. An aperiodic-based approach provides a more general solution to the problem of time-driven resource management that does not suffer from the periodic-based approach’s lack of flexibility. With an aperiodic-based approach, periodic activities can be expressed and handled more generally and directly. For example, sensors can be read, actuators updated, and displays refreshed at specific time intervals (within a specified tolerance). Each time interval occurrence can be managed individually (like loop unrolling) as though it were one more of an unrelated series aperiodic events. Such an aperiodic-based approach meets the demands of aperiodic events as well as exhibiting significant advantages in dealing with the elasticity and evolutionary needs of periodic activities. While periodic-based approaches are the exclusive focus of both practice and theory today, the Alpha operating system departs from this historical trend and represents the first step in the development of the alternate (more promising), aperiodic-based technology.

5 Technical Approach

Alpha was designed to meet requirements defined in the previous chapter, without suffering from the shortcomings of existing systems. This research was directed toward the synthesis of new concepts to meet the particular needs of the previously defined application domain. This required a carefully integrated software/hardware approach, along with the exploration of the tradeoffs required to implement those concepts effectively. One of the results of this research was the creation of a set of programming abstractions that are intrinsically well suited to modular, reliable, decentralized operating systems, and the design and implementation of a set of kernel-level mechanisms in support of them. Alpha is not a monocentric system, that features 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 for the application domain and implemented in concert with each other, resulting in a well-integrated solution within a meaningful framework.

Alpha was designed to be executed on loosely-coupled architectures, with either uniprocessor or multiprocessor nodes. Additional constraints placed on the implementation of Alpha include the requirement that the system 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 the requirement that the system's (native) programming model be similar in concept and implementation to traditional models (i.e., the system cannot force its programmer's to learn and adopt a radically different approach to programming).

Alpha's kernel provides a small, simple set of mechanisms, out of which the higher levels of the system, and the applications, are constructed. These mechanisms permit the exploration of a wide range of system policies, and provide great flexibility for implementing other programming models.

It should be noted here that the Alpha project is directed towards the development of operating systems technology and, as such, the emphasis has been on providing a set of system mechanisms to meet the requirements defined earlier. The intent is that multiple, differing, programming interfaces can be supported by Alpha, ranging from elaborate distributed real-time programming environments, to (more or less) standard object-oriented programming languages, to some standard programming language with direct access to the native system interface. The implementation work on Alpha has been proceeding bottom-up, with the kernel being complete (and somewhat stable) at present, and there is currently no object-oriented programming language (standard or otherwise) implemented on top of the kernel. Instead, the programming experience with Alpha to date has been performed directly on the kernel interface, using a preprocessor to provide the programmer with a set of simple language constructs for programming with the native abstractions provided by the kernel of Alpha [Shipman 88]. Throughout the remainder of this report, the programming abstractions and features of Alpha are discussed from the perspective of programming the native, kernel interface of the Alpha operating system (despite the fact that this is not intended to represent the ideal, or only, programming interface provided by Alpha).

The system's basic programming abstractions were designed explicitly to support the requirements of timeliness, distribution, robustness, and adaptability. In the following sections, the basic abstractions provided by Alpha are defined, the means by which clients of the kernel specify time constraints (i.e., the notion of time-value functions) is described, and the structure of the system's software and hardware is briefly outlined.

5.1 Basic Abstractions

The Alpha kernel interface presents its clients with 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 mechanisms are carefully and deliberately devoid of 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) intended to be the same interface presented by the operating system to an application programmer.

Alpha's kernel is based on a small set of basic mechanisms, in the same spirit as 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.

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 defined to be the manifestations of control activity (i.e., the units of concurrent computation and scheduling) within the kernel.

5.1.1 Objects

A wide range of benefits are claimed for the object model of programming, including increased modularity, the separation of specification from implementation, and the increased reusability of software components. These claims are accepted as true in the Alpha project, and a general discussion of the merits of object-oriented programming can be found in [Cox 86].

In addition to the attributes of modularity, information-hiding, maintainability, etc. normally associated with an object-oriented programming paradigm, the programming model described here is especially well suited for the support of decentralized, high-concurrency implementations of the major robustness techniques supported by Alpha (i.e., atomic transactions and replication).

At the highest level of abstraction, objects in the Alpha kernel are 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 the programmer as individual modules, composed of the object's data and operations that define its interface (similar to Ada packages [Ada 83]). An example object is shown, schematically, in Figure 7, illustrating the data encapsulated by the object and the operations that make up the object's interface.

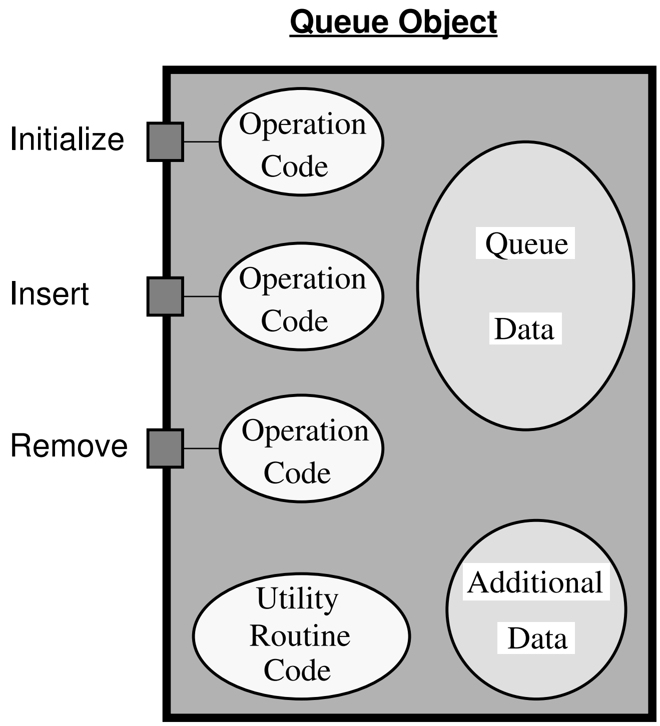

Figure 7: Example Object

Object *types* are static definitions of the code and the initialized data that go into making up an object. Object types are passive entities, defined by the programmers and maintained within the system object store. Object *instances* are the executable, run-time manifestations of objects that are dynamically instantiated from given object types, and may likewise be dynamically deleted from the system. In the following discussion (except where explicitly noted otherwise), the term *object* refers to an instance of a specific object type.

The basic abstractions of Alpha were developed to meet the system's requirements and therefore the definition of objects in Alpha differs from the more commonly accepted definitions. For this reason, the names of the Alpha programming entities were explicitly chosen to carry as little as possible of the semantic baggage associated existing object-oriented programming terminology (e.g., messages, methods, and classes).

In Alpha, objects are simple, passive entities, the main aspects 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, each object in Alpha exists in a separate, hardware enforced address space, and 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 specified by the object's exported operations.

In Alpha, objects are defined by the programmer-in-the-small, as well as the programmer-in-the-large. Any notion of inheritance is assumed in Alpha to be handled at compile-time; there are no specific features of the kernel's interface that are meant to support any form of inheritance.

The object model used in Alpha emphasizes a simple and uniform system interface, minimizing the specialized artifacts that are introduced into the (logical as well as physical) programming model. Alpha supports a quite uniform system programming model—everything appears as objects to the programmer. The object abstraction in the Alpha kernel extends to all system services, and encapsulates all of the system's physical resources, providing clients with object interfaces to all system-managed resources (i.e., memory, devices, etc.). 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 may exist only at a single node at a time; however, objects may be dynamically migrated 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 "system" and "application" objects) is imposed by the programmer, and enforced by the kernel.

The objects in Alpha are expected to be medium to large in size (i.e., on the order of 100-10,000 lines of code). This assumption is derived from some practical considerations having to do with the distributed nature of Alpha and the overhead associated with both inter- and intra-node communications (i.e., in order for the system to be practical, the overhead associated with performing operations must be a small percentage of the cost of actually performing the function associated with the operation).

5.1.2 Threads

Threads represent loci of execution control that move through objects via operation invocations. 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 without regard for the physical node boundaries of the system.

Figure 8 is a snapshot of an example application running on Alpha, consisting of three separate threads in the process of moving through four different objects in the course of performing their computations. Note that the boundaries presented by physical nodes do not appear in this logical view, and both **Thread**$_a$ and **Thread**$_b$ are simultaneously active in **Object**$_3$.

In Alpha, objects are passive entities while threads are the run-time manifestations of concurrent computations in the system—they are the units of activity, concurrency, and schedulability in Alpha. Threads execute asynchronously with respect to each other, allowing a high degree of concurrency to be achieved, but necessitating that the kernel provide a set of concurrency control mechanisms. These mechanisms allow the necessary degree of concurrency control to be applied at a reasonable cost in terms of overall system performance (i.e., the mechanisms perform their functions quickly and their use does not seriously restrict concurrency in the applications).

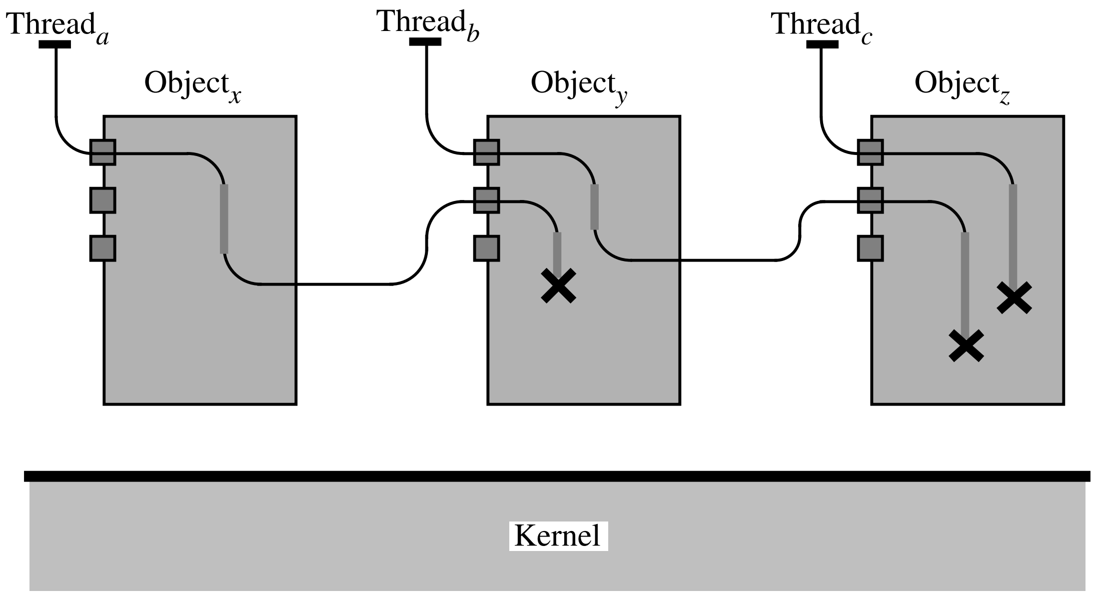

Figure 8: Example Thread/Object Snapshot

The system does not impose an *a priori* limitation on the number of threads that can be executing within an object at any given time. Should concurrency control be required, it is either up to the system level programming interface or the applications programmer to apply concurrency control restrictions (either in a brute-force, or application-specific fashion). However, the kernel provides the necessary (basic) concurrency control mechanisms, and once specified, the system will enforce the desired restrictions on the concurrent execution of threads. In addition, this approach ensures that the thread concept extends naturally to the multiprocessor case (i.e., with concurrency within nodes, as well as among them).

With threads it is possible to implement a wide range of system-level control policies, ranging from low-concurrency structures (such as *monitors*) 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. 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 (for further details, see [Northcutt 88a]).

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.

Threads include the local state information for the computations that threads represent. This includes attributes related to the nature of the computation (e.g., reliability requirements, timeliness constraints, and relative importance) that can be used by the system to more effectively manage the system's resources. 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 executes through objects, typically in a nested fashion as represented by the thread in Figure 9, which acquires new attributes in the course of its execution (shown both in snapshot and “straightened-out” form).

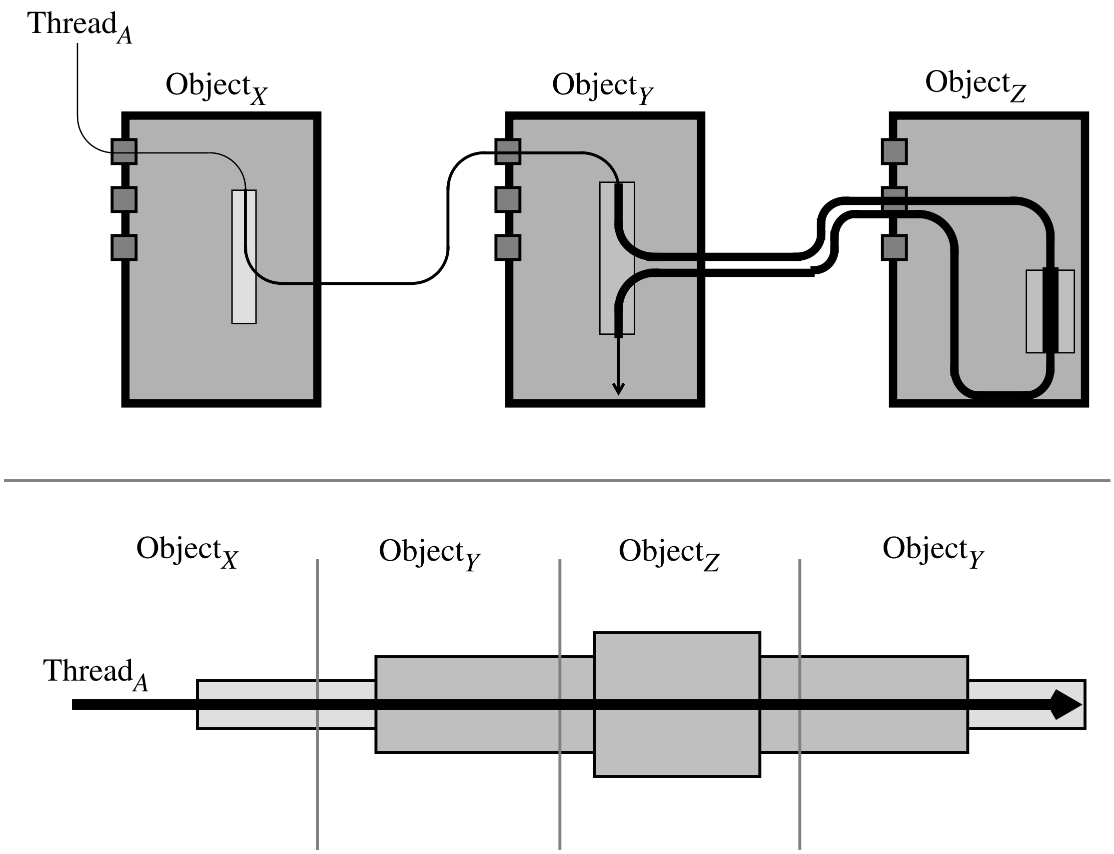

Figure 9: Example of Thread Attribute Nesting

Threads are independent of any specific object, they move among objects, providing animation for these otherwise passive entities, but bear no special association with any object in the system. Furthermore, threads can be dynamically created and deleted in the course of an application’s execution. Each time a new concurrent activity is to be performed, a new thread can be created. With Alpha, threads are the form taken by all activity in the system.

From a conceptual standpoint, the Alpha notions of thread and object cleave the point of execution control from code and data definitions in the standard process notion. 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.

5.1.3 Operation Invocation

The variety of objects found in Alpha interact most naturally through a form of communication characterized by send/wait interprocess communication. Objects in Alpha therefore use a type of Remote Procedure Call (RPC) for the invocation of operations on objects (as illustrated in Figure 10).

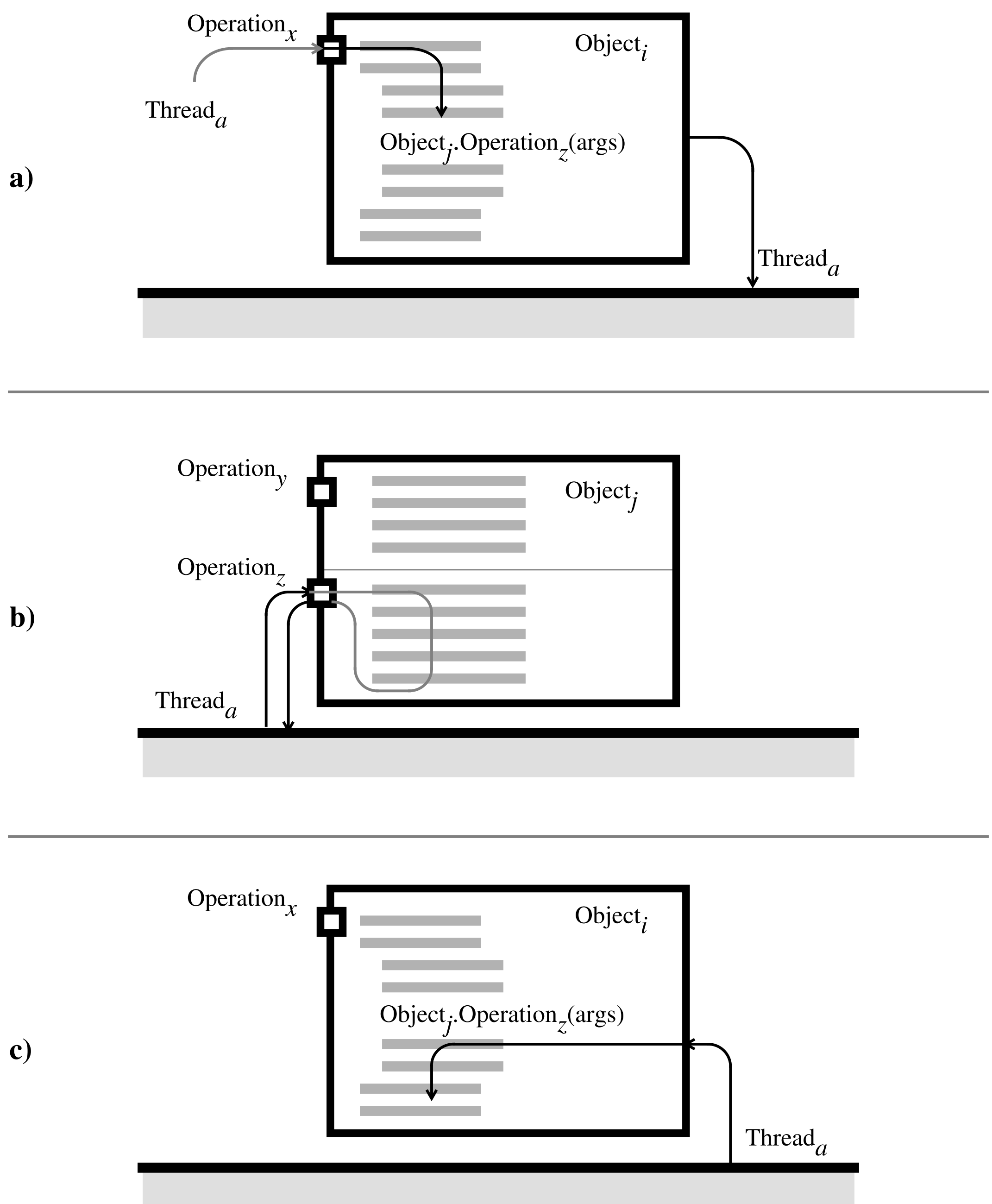

Figure 10: Example Operation Invocation

All interactions with both user and system objects is via 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 facility provides simple, uniform access to all objects, whether local or remote, and provides reliable RPC-like semantics. Each invocation returns a success/failure indication (with error code for failures) on its completion. The operation invocation facility does much to mask the undesirable effects of the system's physical distribution (i.e., node failure, message errors, non-local objects, object migration, etc.), and provides time-driven orphan detection and elimination.

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. 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.

The operation invocation facility was designed to retain as much of the familiar subroutine-call semantics as possible. The operation invocation facility allows the passing of arguments among objects, with return/value semantics. It is possible to pass simple, or structured data, as well as capabilities as arguments in operation invocations.

Because of the fact that all of Alpha's system services are provided in the form of objects, the operation invocation facility subsumes the role of traditional "system calls," and it constitutes the single trap entry point into the system.

This form of thread/object interaction has a number of good features, including the fact that 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]. Also, the use of a synchronous form of communication in Alpha does not pose the types of limitations that are typically associated with synchronous communications in process/message-based systems [Liskov 85], due to the nature of the object and thread abstractions. Furthermore, there was initially reason to believe that this type of system could be implemented in a way that is suitably efficient to make possible the construction of meaningful applications programs.

5.2 Time-Value Functions

Alpha uses a novel and highly effective technique for explicitly and expressively manifesting an application's time constraints—as the time-dependent value to the system of completing the computation's activity. 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).

The attribute information that is associated with threads in the current implementation of Alpha include: an indication of the relative importance of the thread, the expected completion time for the execution of a region of code; the value (with respect to time) to the system of completing the execution of a section of code; and a probability distribution function that indicates the probability (with respect to time) of completing a region of code. These attributes are associated with threads by parameters specified when creating new threads, and may be modified (at run-time) by invoking operations on the threads themselves.

Collectively, these attributes are the Alpha system's manifestation of *time-value functions*, which represent the value to the system of the execution of a computation (or parts thereof) with respect to time. Time-value functions distinguish between a computation's timeliness and its importance, it is an aperiodic-based, dynamic approach that is both very powerful and expressive. Time-value functions consist of several different components, as illustrated in Figure 11. The concept of time-value functions was invented by Jensen in 1975 [Jensen 75] and was explored by one of his students working on the Archons project [Locke 86].

$\boldsymbol{I}$	— 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

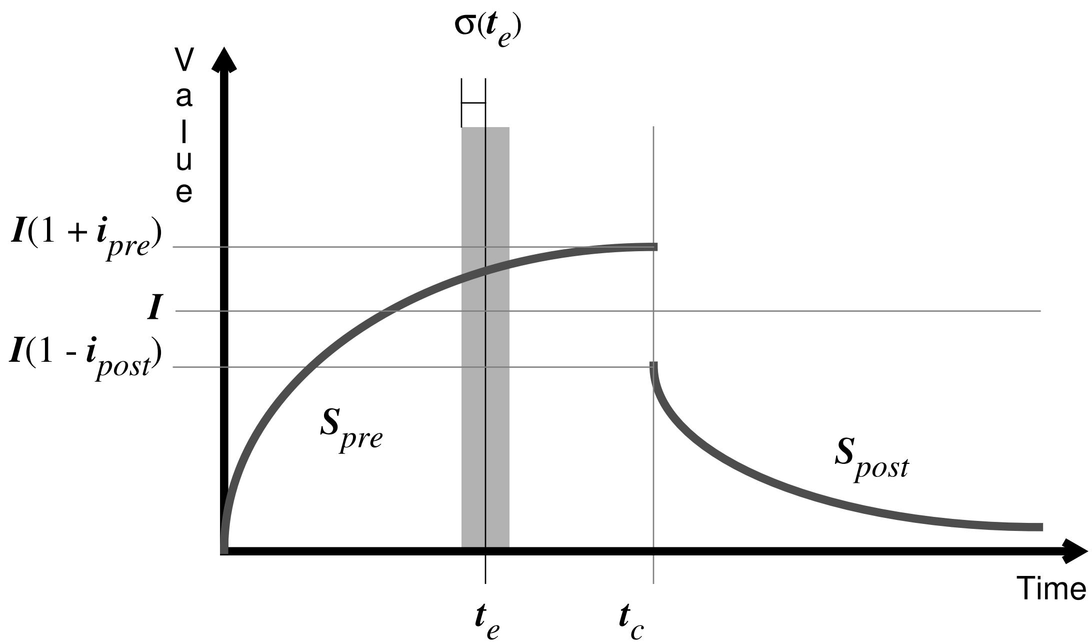

Figure 11: Components of a Time-Value Function

Time-value functions are the basis for resolving all contention for resources in Alpha—e.g., processor cycles; communication management; secondary storage access; synchronization resources (e.g., semaphores). The user-specified time-value functions and the run-time statistics maintained by the system for each thread (such as 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 the same general set of guidelines. The time-value functions for all contending activities are evaluated *collectively,* and then the activities are scheduled so as to maximize the total value to the system for the entire time span they cover. The urgency of the computations requesting system resources is considered first—when there sufficient resources to do so, all of the resource requests are serviced in such an order as to ensure that the computations' time constraints are all 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 [Jensen 76a]). 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. The currently preferred policy in Alpha takes the former—i.e., contending requests are selectively denied, on the basis of the urgency and importance of the computations responsible for the resource requests. Further details on this type of resource management is given in Subsection 6.1.3 of this report.

While it is true that higher-utilization techniques impose a performance cost of their own, our studies have shown that in most cases this cost is worth the benefits that it provides [Locke 86]. In fact, it has been shown that many time constraints may not be met under conditions of extremely low utilization [Lechovski 84]. The results of these efforts have illustrated that a large component of the cost can be offset through the use of hardware currency—i.e., a simple, special-purpose hardware accelerator (analogous to a floating-point co-processor). The Archons project's analysis, simulation, and experience indicate that the resources consumed in performing time-value resource management produce greater value for the system than do the simpler techniques which consume considerably smaller amounts of resources that could otherwise be used by the applications. In any event, this project is interested in exploring this research direction in an attempt to develop operating system techniques that will yield both increased system resource utilization and improvements greater than the normal hardware-technology-driven performance increases.

5.3 Implementation Features

The following subsections describe the structure of the software and hardware in the initial implementation of Alpha .

5.3.1 System Software Structure

The Alpha operating system lies in between the application code and the hardware and (in its current implementation) has an internal (logical) structure as shown in Figure 12. The bulk of the functionality described in the preceding sections is provided by the kernel layer, and it is the kernel layer that has received the greatest amount of attention by the project to date. Future work will be directed towards enhancement of the executive and system layers. The following provides a brief description of each of the (logical) software layers in Alpha, including the *Application Layer, System Layer*, *Executive Layer*, *Kernel Layer*, and *Monitor Layer*.

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, much as 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, as defined by the capabilities that they possess. 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.

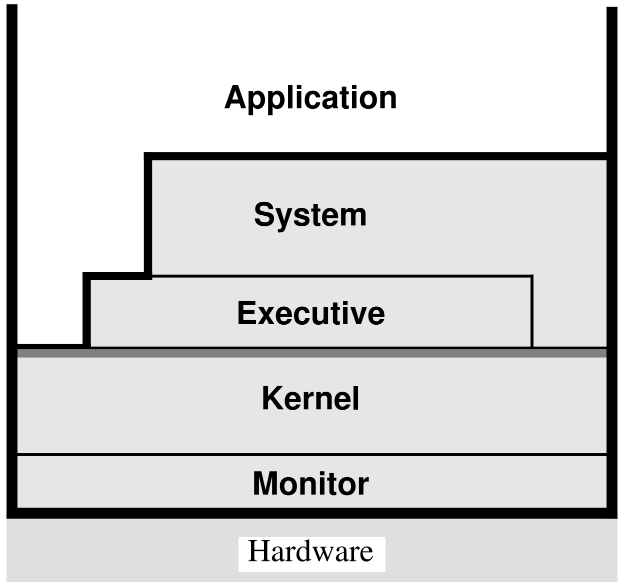

Figure 12: Logical System Software Structure

The executive layer extends the functionality of the kernel by way 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 are have special characteristics and are implemented in a different manner. Kernel threads and kernel objects are 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. Kernel object types are specified the same as with normal objects, however they are linked into the system at system-build time and kernel object instances coexist within the kernel's address space. Kernel threads and objects are used to implement system daemons (e.g., interrupt handlers, virtual memory pager, transaction manager, and garbage collector). 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 provide services to the client by way of object interfaces (e.g., object and thread managers, and semaphore and lock managers)
- hardware supported 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., operation invocation, trap handling, subsystem interfaces, and physical memory and I/O management)

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

5.3.2 Testbed Architecture

The Alpha operating system executes on a loosely-coupled collection of dedicated-function multiprocessor nodes, constructed from largely off-the-shelf system components (i.e., the Alpha Distributed Computer Testbed [Clark 83, Northcutt 88b]). The testbed supports the development of software by a collection of system programmers, working from individual (remote) workstations. Furthermore, the nodes of the testbed were designed to 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 13). The development and control system consists of a collection of Sun Microsystems workstations, running the UNIX operating system, and are connected to the distributed computer system via both Ethernet and 9600 baud serial lines. This part of the testbed is used for the development of application programs, the control of applications running on the distributed computer system, and the monitoring of the operating system as well as application experiments. The software tools that exist on the Sun Microsystems workstations (editor, compiler, linker, window manager, etc.) provide an effective development environment, permitting the testing and debugging of low-level system code by multiple, remote users.

The distributed computer system component of the testbed is the host on which the Alpha operating system executes. It consists of a collection of processing nodes interconnected by a global communications subnetwork. This system is (logically) a single computer, where the network is analogous to the backplane and the nodes being the cards of a conventional processor. 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 on which the Alpha kernel executes, 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 can be a combination of actual or simulated devices. The application system interfaces to the distributed computer system directly through nodes, the network, or gateway machines. Details concerning the Alpha Distributed Computer Testbed can be found in [Northcutt 88b].

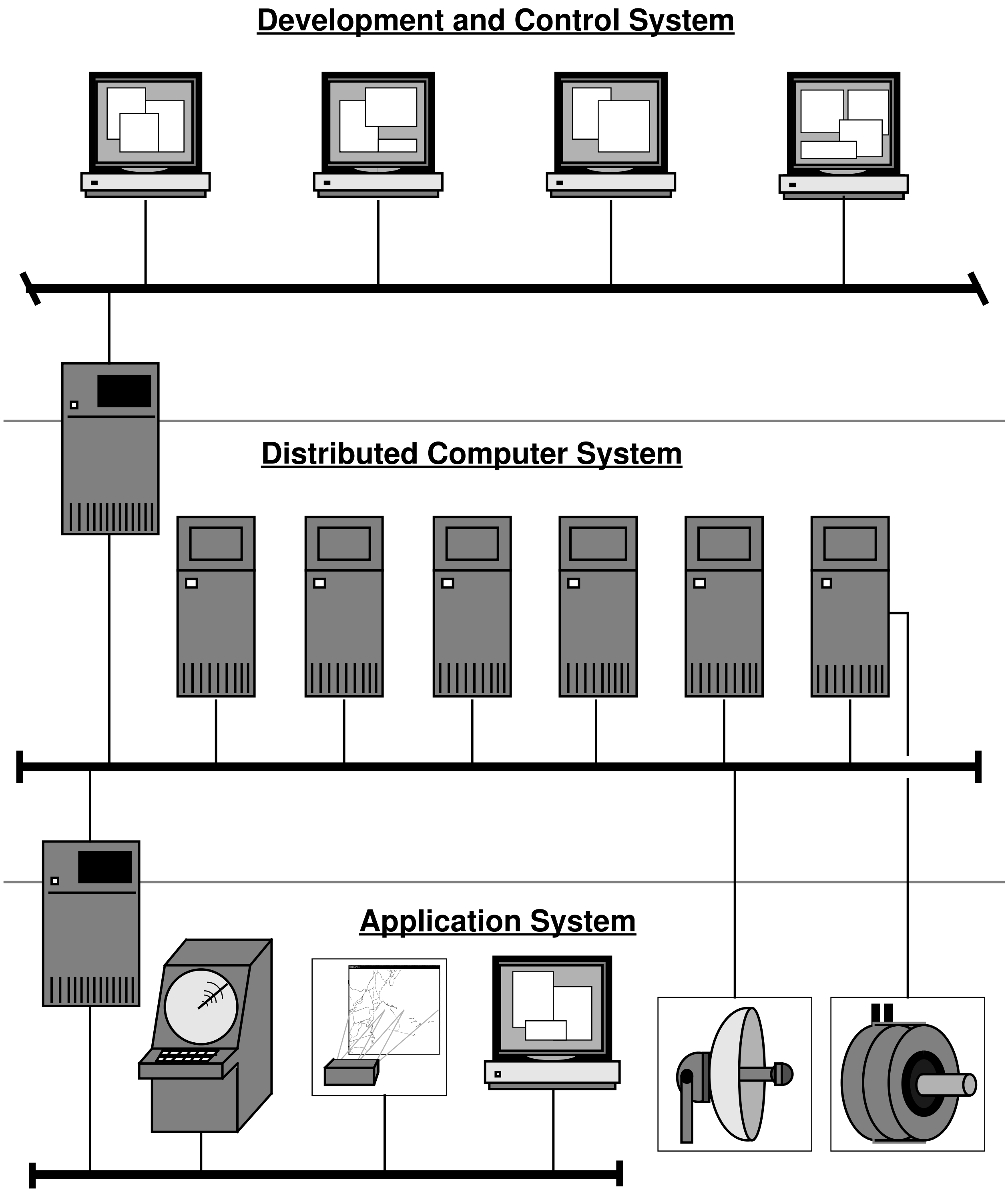

Figure 13: Testbed System Structure

6 Rationale

The following provides an overview of how each of the major issues associated with distributed real-time control systems is addressed by Alpha's kernel mechanisms developed as part of this work.

6.1 Timeliness

Because of the importance of timeliness issues, much attention has been given to considerations of time in the specification, design, and implementation of the mechanisms that make up the Alpha kernel. Most of the common techniques for meeting the (simpler) requirements of real-time systems have been applied in the design and implementation of the Alpha operating system. In addition, each of the facilities in Alpha was specifically designed to help meet the requirements of real-time applications. The general approach to timeliness taken in Alpha is highly adaptive, permits high utilization of system resources, and addresses the problems of timeliness head-on, without making unrealistic or limiting assumptions about periodicity or determinism.

The most visible manifestations of the concern for timeliness in the design and implementation of Alpha are: time-driven resource management, and the use of application-specified time constraints and resource management policies. As defined earlier, the aspect of an operating system design that is most responsible for meeting the needs of real-time applications is the manner in which contention for system resources is resolved; real-time operating systems should take into account timeliness constraints when resolving contention for resources. 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 the requests cannot be met in a timely fashion, an application-specific overload handling policy is followed in dealing with the requests. To this end, the thread abstraction was developed to provide the framework within which global, time-driven resource management can be performed.

An implication of the use of time-driven resource management in Alpha is that the client must provide the system with a policy for managing its resources (both in normal and overload cases—i.e., when there are insufficient resources to meet the application's needs). In addition, the client must provide the system with the information needed to manage each of the application's time-critical computations (e.g., the timeliness constraints and relative importance of each computation). The Alpha operating system was designed to allow the system's clients to provide custom resource management policies (e.g., for application processor scheduling and virtual memory paging), and to allow the simple and natural expression of application-specific time constraints (by associating timeliness attributes with threads).

The influence that the requirements of real-time programming had on the design and implementation of Alpha is evident in the system's basic abstractions, programming interface, kernel subsystems, and kernel mechanisms.

6.1.1 Effects on the Basic Abstractions

The Alpha fundamental programming abstractions were specifically designed to support the overall objective 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.

6.1.1.1 The Thread/Object Approach

Threads maintain a direct association between 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, location-transparent, 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.

In Alpha, all requests for system resources can be associated with the computation that made each request, and the system uses these attributes to service the requests according to the application-specified management policy for that resource. For example, the current scheduling policy in Alpha sequences the execution of threads in such a fashion as to allow all of the threads' specified time constraints to be met (whenever possible), using the (time-varying) urgency and importance attributes of threads that are ready to run.

The thread abstraction in Alpha was not designed to meet the requirements of timeliness in isolation, but the requirements of distribution, robustness, and adaptability were also considered in the design. 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 (i.e., at each node, the same resource management policies are applied to the threads' global attribute information).

The Alpha thread/object abstractions are better suited for meeting the needs of distributed real-time control systems than the conventional process/message abstraction. The main reason for this is that the Alpha thread/object abstraction maintains a strong correspondence between the logical and physical views of computations, while typical process/message-based systems do not maintain such a strong relationship. This close association in Alpha allows requests for system resources to be tied to the logical computations (that the programmer associated attributes, such as time constraints, with) to allow the time-driven management of resources.

6.1.1.2 Comparisons to Traditional Approaches

To illustrate some of the significant differences between Alpha and traditional process/message-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). In most systems, the functionality of an application is decomposed into software modules for reasons of intellectual manageability, reusability, relocatability, and concurrency. In the process/message approach, the software modules that are combined to make up computations are known as *processes*, while in Alpha, the programming modules that define computations are known as *objects*. In a normal process/message-based system, these logical computations are mapped onto collections of processes that interact by message exchanges, while in Alpha each computation is defined by a thread that moves among a set of objects via operation invocation.

Figure 14 illustrates a stylized process/message-based implementation of this example application, where processes *1*, *2*, and *3* obtain the service provided by process *x* through the exchange of messages. Figure 15 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 discontinuity between the programmer's logical concept of three application computations, and the four interacting processes that the system supports in its implementation. This discontinuity requires that the programmer perform a transformational step in going from the application's conceptual design to its 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 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 14, 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. The result of this is that process *x* executes at its given priority level, regardless of on who's behalf it is performing its function, so the portion of the first computation being provided by process *x* is executed at too low a level of priority, and similarly part of third computation runs at a higher level of priority than it should. This example, and various schemes that have been developed to help processes deal with these problems (e.g., priority propagation) are dealt with in Chapter 6.

In Alpha, each thread represents an individual computation and moves among objects independently, 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 in each of the threads in Figure 15 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 a given time. In effect, 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 intermodule communication. In typical process/message-based systems, communication and scheduling activities are intertwined in an undesirable fashion— sdasdfto

Process	**Priority**
1	High
2	Medium
3	Low

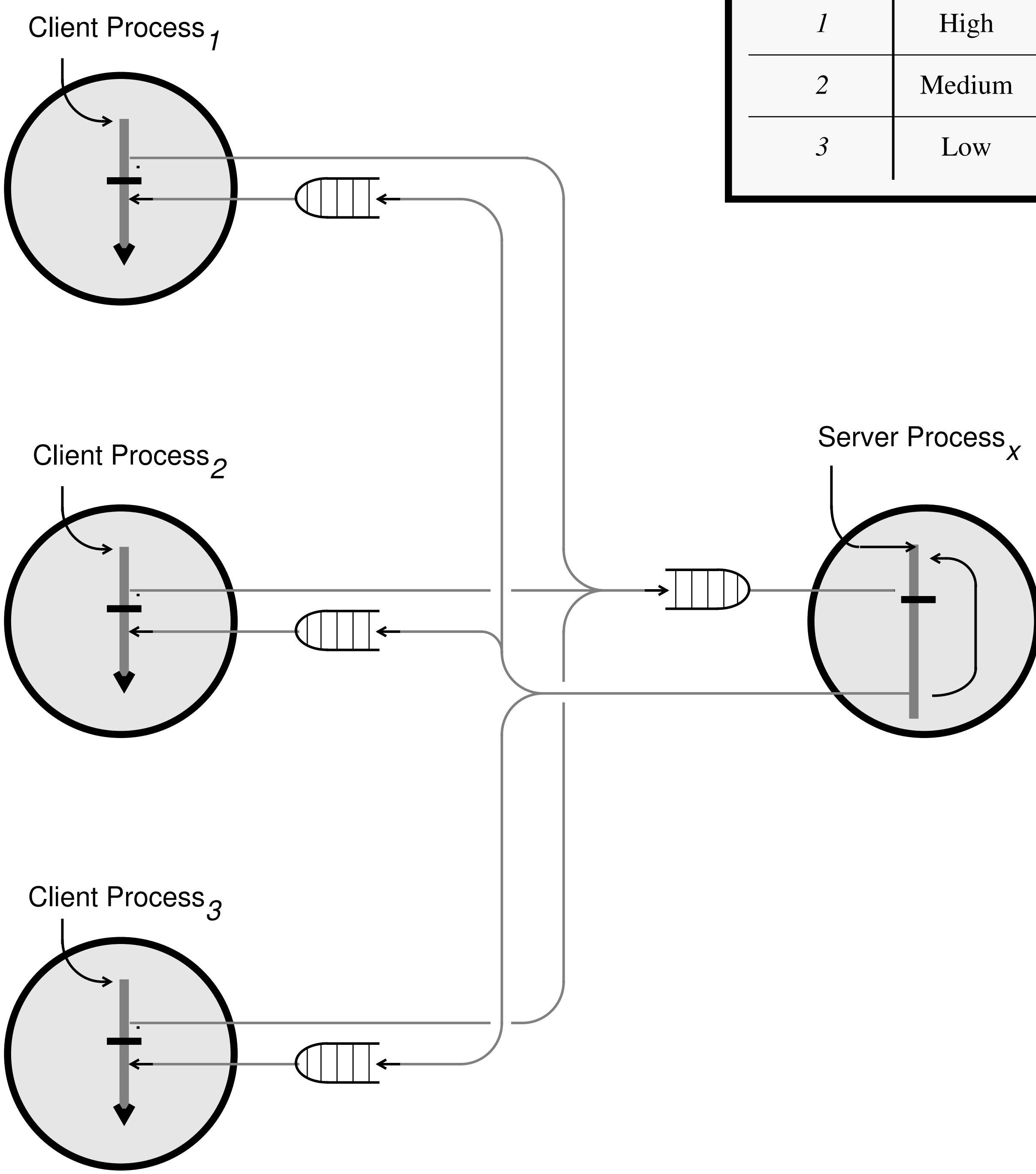

Figure 14: Typical Process/Message Interactions

Thread	**Urgency/ Importance**
a	High
b	Medium
c	Low

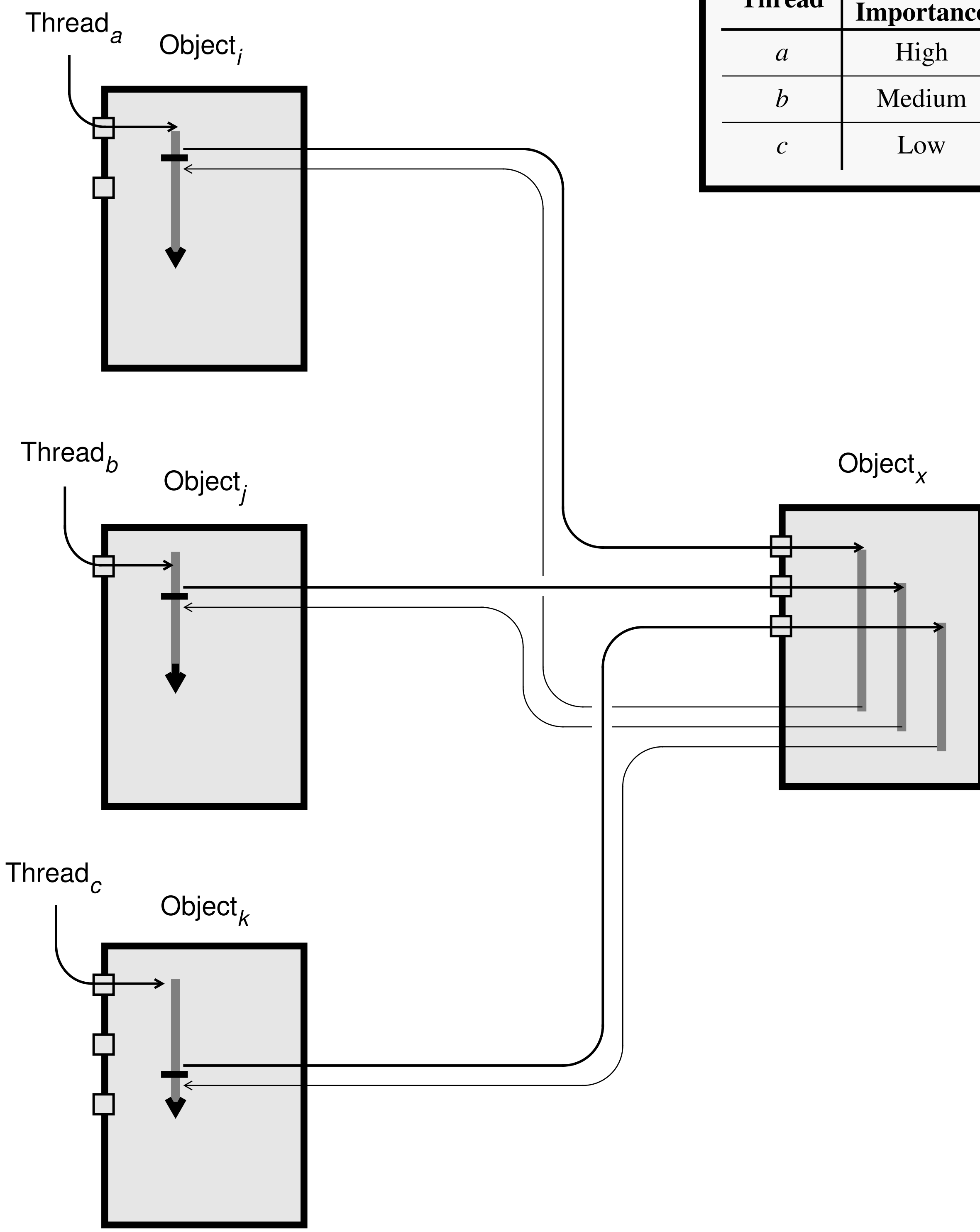

Figure 15: Typical Thread/Object Interactions

to accomplish the next (logical) phase of a computation requires a scheduling activity. This is an artifact of the process/message approach, because communications itself is not 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., the threads), while in the process/message example there are multiple schedulable entities involved (i.e., the processes).

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, communications, or memory resources required by the 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.

6.1.2 Effects on the Programming Interface

The Alpha system's timeliness requirements have had a very dramatic effect on the system's programming interface (i.e., the interface that the kernel provides to its clients). In particular, the use of time-driven resource management demands that the programmer be able to provide the system with the information needed to carry out the desired resource management policy. Furthermore, the very fact that time constraint attributes may be specified for computations implies a need for the programmer to be able to deal with the effects of missing hard deadlines (i.e., time constraints that define a time after which the computation no longer has positive value to the system).

6.1.2.1 Specifying Timeliness Attributes

The Alpha programming interface supports the notion of *time constraint blocks*, that provide a convenient way for the programmer to define time constraints for the execution of specific regions of code within objects. Time constraint blocks allow the programmer to associate dynamic time constraints with blocks of code by, in effect, placing brackets around a region of code. Alpha supports the application of multiple, nested time constraints to threads, where all of the time constraint blocks in which a thread is active apply at all times (i.e., a nested constraint does not invalidate those enclosing it).

Alpha provides the programmer with flexibility in determining the binding time for 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 enclosed by 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 a block of code has a time constraint, but exactly what the constraint is 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 sample past a sensor). Furthermore, there are different types of late binding that can be performed. The parameters for a time constraint block may be a function of the global state of the object alone, in which case the timeliness 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), in which case the timeliness constraint for the block of code could vary with the thread executing within it. This flexibility allows the programmer to more dynamically and accurately represent the timeliness needs of his computation to the system, thereby making it possible for the system to manage its resources more effectively.

6.1.2.2 Handling Expired Time Constraints

When a thread's time-value function is no longer positive (i.e., the thread has failed to complete the execution of a time constraint block within the specified amount of time), the thread's execution within the time constraint block should be terminated. This is because the thread's time constraint cannot be met, its continued execution and consumption of resources interferes with the successful meeting of other threads' time constraints, and the work that the thread continues to do represents more effort that must be expended to compensate/undo to make the object consistent.

The fact that there may be considerable negative value to the application of the continued execution of threads whose time constraints cannot be met demands that the system must put a stop to the execution of such threads in a prompt fashion. Simply halting a thread when its time constraint has been missed is a totally unacceptable solution—in a real-time system, missing time constraints is as much a part of life as defining and meeting time constraints. A much more meaningful action to take when a time constraint expires (e.g., a deadline is missed) is to redirect the execution of a thread to an exception handler specified by the programmer, where the programmer can define what should be done. The programmer might want to perform one or more of these actions: kill the thread, put the object back into a consistent state, report an exception to another object, compensate for the partial execution of the block, retry the block, log an event, or just proceed 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 (as illustrated in Figure 16). 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., an abort induced when it becomes known to the system 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); 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 a time constraint block. This implies that once a thread's execution is directed to a time constraint block's abort 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 pops to the next level of time constraints so that the thread's attributes properly correspond to the thread's new 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 largely the same way that block-structured, nested, mechanisms are used for handling other types of exceptions in the system (e.g., atomic transactions).

The kernel can also 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.

```
/* 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 16: Example Use of Time Constraint Blocks

6.1.3 Effects on the Kernel Subsystems

Each of the major kernel subsystems (i.e., the scheduling subsystem, the communications subsystem, and the storage management subsystem) was affected by the system's timeliness requirements. The following paragraphs give an overview of the major effects that timeliness had on the design and implementation each of the subsystems.

6.1.3.1 Scheduling Subsystem

The primary system resource that any operating system manages is application processor cycles, and in Alpha this resource is managed by what is known as the scheduling subsystem. The subsystem was designed and implemented to support: a wide range of different scheduling policies, the management of multiple applications processors, and the computation of schedules concurrently with respect to the execution of application code. In the current implementation, the scheduling subsystem executes on a separate processing element within a node, and interacts with the kernel proper (executing on an applications processor) via the kernel's inter-processor message-passing mechanism. In Alpha, per-node scheduling is done at the kernel level, while global scheduling is performed at the system level.

The scheduling subsystem on each node maintains information on the time constraints of all threads currently active on that node, as well as their current state with respect to those constraints (e.g., the thread's accumulated execution time, whether the thread is blocked, and what system resources the thread has acquired the use of). The scheduling subsystem on a node generates a new thread execution ordering (i.e., schedule) on each scheduling event, and causes an applications processor to perform a context swap and dispatch a new thread when the currently executing thread is no longer at the top of the schedule. Scheduling events occur when a thread: is created/destroyed, unblocks, blocks, is run, is preempted, changes importance, enters/exits a time constraint block, handles an exception.

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 policy. The best effort scheduling algorithm is an outgrowth of the Archons project's long-standing research efforts in real-time scheduling and its use in Alpha 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 expected value to the system of the threads that are being scheduled for execution.

The best-effort policy includes a particular overload handling sub-policy, however 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 have all threads be tardy by some average amount of time (i.e., distributed the overload evenly across all ready threads); or do not discard threads, but have the ready threads be tardy by an amount inversely proportional to their value to the system.

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.

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 can 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 combination of time-value functions and 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 deadlines, soft time constraints, hard or soft execution time windows, and delayed execution. Because time-value functions can be specified dynamically (and the time constraint parameters may be run-time variables), a time constraint applied to a 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).

Other, more typical, scheduling algorithms do not attempt to deal with overload conditions properly (in fact, some policies, such as deadline, 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. Many real-time 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. Cyclic events are "unrolled," with each iteration treated as a 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 system does not fall apart when an assumption concerning the periodicity of events is violated, nor does it require a reevaluation of all of an application's timeliness constraints when a change is made in one section of code.

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 their "raw" form and are not transformed into some compressed form such as priorities, meaning 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 its computation, the system can be more adaptive than those which statically define time constraints for computations.

The best-effort scheduler allows the Alpha system to be employed in a more conventional manner that guarantees real-time constraints can be met for low-level static applications (e.g., rate-monotonic or rate-group scheduling). However, Alpha's best-effort scheduler has significantly greater capabilities for more general real-time command and control applications than is available in typical real-time schedulers.

6.1.3.2 Communications Subsystem

The demands of timeliness are also reflected in the design and implementation of the Alpha communications subsystem. This subsystem manages the resources associated with the communications between nodes in the distributed computer system, including interconnection bandwidth and the processing cycles and memory associated with the communications activity. These resources can be managed in the same fashion as the scheduler uses to manage application processor cycles—i.e., trace each request to a thread and use its attributes to help meet the desired time constraints and to resolve conflicts when contention for resources occurs. In this way, the communications resources can be managed in the same fashion as all others, using the same information and the same policies.

The current testbed hardware does not include an interconnection network that uses a priority-based bus arbitration scheme. However, if such a bus were used, this notion could carry the desired resource management policy all the way down to the bus-arbitration level. For example, the time-value information is used to provide the most important activity on each node with access to the bus, the bus arbiter could then be used to allow the most important activity across all of the nodes to use bus next†.

6.1.3.3 Storage Management Subsystem

The storage management subsystem also reflects the timeliness requirements in the manner in which it manages the system's secondary storage and virtual memory resources. The storage management system makes use of the attributes of the threads that are responsible for storage resource requests—e.g., page-in requests, physical memory page allocation requests, and write page to stable storage requests.

The storage management subsystem also maintains the run-time statistics that are required to determine the working set of pages needed by each thread executing within a particular object. The storage management subsystem uses the client-specified attributes of threads in conjunction with its own run-time statistics to manage storage resources (i.e., virtual memory, primary memory, and secondary storage) according to a given policy, just as the scheduling subsystem does for application processor cycles. For example, the virtual memory policies can use application-specific information about individual threads to do pre-paging and page victimization in a timely fashion—e.g., do not victimize pages needed by threads with short deadlines, order the servicing of pre-paging requests based on the time constraints of the requesting threads, and order the disk read and write request queues based on the time constraints of the threads associated with the blocks being read and written.

6.1.4 Effects on the Kernel Mechanisms

The design and implementation of the Alpha kernel's mechanisms provide further support of real-time applications. For example, the synchronization primitives are designed to consider

---

†*Note that this is only an issue when it is necessary to obtain extremely high utilization from the bus and the non-preemptable period of bus usage is significantly larger than that which is typical of the computing speed versus communication speed ratios that exist in products today.*

timeliness constraints in their function; each time a thread performs a synchronization operation (i.e., manipulates a semaphore or a lock), the scheduler is notified. When the time comes to unblock a thread waiting for a synchronization token (e.g., when a V operation is done on a semaphore, or when a lock is released), the timeliness attributes of each of the blocked threads is considered and the thread with the most demanding time constraint is chosen. 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 is desirable to allow a thread with less stringent timing constraints to execute before one with tighter constraints in order to allow free up a thread with even greater urgency that is waiting for a synchronization token to be released (for examples of this, see [Northcutt 88a]).

The Alpha operating system was designed so that the execution of threads can be preempted while they are executing within the kernel. This is as opposed to the case in some operating systems (e.g., UNIX) that disable preemption within the kernel. This is done to reduce the amount and complexity of synchronization required by the system to maintain the consistency of its internal data structures. However, an application typically spends 40%-50% of its time executing within the operating system's kernel, and so a system's responsiveness to the dynamics of real-time applications suffers if preemption is not permitted while a computation's point of control is within the kernel.

Another example of the influence that the system's timeliness requirements had on the design and implementation of Alpha is that (to the greatest extent possible) all system activities are implemented with threads. This was done because of the observation that there are many activities in typical operating systems that fall outside the realm of normal, client computations, and as such are not governed by the system's normal resource management policies. For example, interrupts, DMA-type activities, and (hardware) exception handling are all activities that consume (or "steal") system resources (e.g., processor cycles, memory, and I/O bandwidth), but are usually not managed by the operating system. By converting all such activities into threads (e.g., all interrupts are converted almost instantly into threads—buffer data, reset device, and unblock a thread), the system can manage more of its resources according to their given time constraints, in a uniform and globally consistent fashion, according to the application-defined policy.

In addition, the Alpha kernel mechanisms were designed and implemented so as to ensure that they will not require highly variable or unbounded amounts of time to complete their functions. This is meant to enhance the predictability of behavior, as is frequently desired of real-time systems. Another aspect of this is manifest in the application of optimizations that were directed towards the exception rather than the normal cases (examples of which are provided in the following sections).

6.2 Distribution

One of the basic assumptions for Alpha was that it would execute on a physically distributed computing system, for reasons of both robustness and performance. A distributed system can provide concurrency of execution among (as well as within) its constituent nodes, and the bottlenecks of traditional centralized systems can be avoided. Distributed computer systems mirror the physical distribution of real-time command and control applications by placing processing elements in near proximity to the sources and sinks of application information. This results in a reduction in communications delays between processing and data source/sink sites, and allows the specialization of processing nodes for increased overall system cost-effectiveness. Physical distribution of a computing system can also improve its survivability by providing independent failure modes, greater isolation of faults/damage, and improved support for graceful degradation of function and dynamic reconfiguration.

To obtain the benefits offered by distributed systems, a number of additional functions must be performed and additional features must be dealt with that either didn't exist in centralized systems or could be safely ignored. Either the system software or the application programmer must perform inter-node resource management and deal with the effects of physical distribution. Should this be done by the application programmer, the effort is recurring, possibly non-standard, and frequently non-expertly implemented. It is furthermore difficult, or impossible, to coordinate the resource management functions performed by the clients with that (largely local functionality) provided by the underlying system software.

The Alpha operating system deals explicitly with the effects of distribution by means of 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.

The kernel provides mechanisms to permit the programmer to deal with node and communications network failures (i.e., the operation invocation facility), and to dynamically (and transparently) migrate objects among nodes. 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.

The Alpha system abstractions do not directly reflect the (significant) performance differences in interactions between objects co-located on a node, and objects on different nodes. Whereas some operating systems introduce sub/super-structure abstractions that recognize and exploit the differences between local and remote objects, in Alpha the intent was to maintain a uniform programming abstraction that projects to the programmer an image of a (logically) centralized system. Should additional structure among the programming entities be desired, it can be applied at the system or applications level. This approach was taken with the desire that the system software abstract away the undesirable aspects of physical distribution, and the kernel-provided programming abstractions maintain a strong relationship to their implementation. In some systems, there are sub-computational entities that are a part of the programmer's abstractions but are not visible to the underlying system software. This approach suffers from the fact that there is a loss of correspondence between the logical and physical aspects of computations; the system manages a single entity, which encompasses more than one of the programmer's computations. In such cases, the programmer's individual computations cannot be independently relocated across the distributed system at run-time. A further implication of an approach where the programmers' abstractions differ from their system implementation is that the system is unable to distinguish among the requests for resources it receives and so it cannot perform its management functions effectively (e.g., when one sub-entity generates a page-fault, all of the related entities are blocked).

6.3 Robustness

For the purposes of this work, a commonly used distributed system failure model was adopted [Anderson 81]: the types of failures considered here are the failures of both hardware and software components, 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 support (independently) 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 operating system 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 the kernel for atomic transactions and replication are initial versions of the more complete sets of mechanisms being developed in these on-going projects. The on-going work by the Archons project is addressing the development of system-level policies which use these mechanisms. In fact, the area of real-time atomic transactions is addressed by an in-progress thesis research project [Clark 88].

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 88, Sha 85]. The Alpha object model provides a simplified (or constrained) control structure for interactions among software components (as compared to a more general process and message-based system model) and restricts the accessibility of the 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 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).

6.3.1 Exception Handling

The different types of exceptions that can occur in Alpha are classified into three groups: *client-defined*, *kernel-defined*, and *machine-defined* exceptions. Client-defined exceptions indicate client-level events by way of the system's normal operation invocation facilities. For example, the failure of a client operation may be indicated to the invoking object by way of an operation invocation return parameter. Also, objects can "register" operations with other objects that are to be invoked when a given client-level exception occurs. In this way, a thread can invoke a given operation on an object when a client-level exception is detected, which may change the state of the object in such a way as to allow other threads to detect the exception (i.e., an implementing an interrogative signal, as opposed to an imperative one). These exceptions exist at the application-level and system-level in Alpha and the kernel is typically not involved in the signalling or interpretation of this class of exceptions. However, the application programmer is capable of generating other types of exceptions with his code—e.g., divide by zero or invoke an abort operation. Even though an explicitly generated operation, like a transaction abort, is not strictly an exception, its effects are the same and it is treated as an exception in Alpha.

System-defined exceptions stem from the operating system's fundamental behavior and can be classified into three groups, depending on which system feature is responsible for the exception. The system-defined exceptions are due to the fact that Alpha is a distributed operating system that supports time constraints and atomic transactions. 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 detached (i.e., *orphaned*) computations to be created. Therefore, the system must provide a means of indicating the failure of operation invocations, as well as a means for detecting and eliminating orphan threads. In addition, it is necessary that the system detect and eliminate orphaned portions of threads in a timely fashion. The longer that an orphaned computation continues to execute, the more resources it wastes (because such a computation is not able to complete, therefore the work it accomplishes is in vain), and the greater the amount of effort that will have to be expended to compensate for the actions it has performed. This means that an asynchronous thread section abortion mechanism is called for, to eliminate portions of threads regardless of whether they are blocked waiting on a remote invoke or are actively computing (i.e., either the head segment or body segments). This asynchronous thread abort mechanism must accomplish the orderly repair of "broken" threads (i.e., notify the new head of the thread when the orphaned section(s) have been successfully eliminated), and it must also be compatible with time-driven resource management objectives and mechanisms.

Any operating system that purports to support the time-driven management of resources must provide a means of dealing with the exceptions which stem from the inability of a computation to meet its given execution time constraints. The exceptions which occur when a time constraint expires require the same prompt attention as is needed for orphaned thread detection elimination. By definition, the continued execution of a computation that has missed a deadline (i.e., a hard time constraint) can no longer provide value to the system (and may in fact be counter-productive to the system). Therefore, an asynchronous exception mechanism is called for in this case as well. The system should provide a means of diverting control from a thread's normal flow of execution to an exception handler whenever a time limit (as defined by a thread's timeliness attributes) expires. Furthermore, this exception handling mechanism should preserve the state of the threads that miss time constraints, in order to permit the graceful recovery of a computation following an expired time constraint. As with all mechanisms in Alpha, this exception mechanism must work in concert with the system's overall time-driven resource management philosophy. The use of atomic transactions also imposes a requirement that the system support the (both synchronous and) asynchronous diversion of a thread's control that results when a transaction aborts (e.g., because of node failures or as a result of an explicit transaction abort command). As far as mechanisms for manipulating threads as a result of exceptions are concerned, the needs of atomic transactions are very similar to those of handling thread breaks or missed deadlines.

Machine-defined exceptions stem from events in the underlying hardware (e.g., divide by zero, bus error, non-existent memory, and access violations). All operating systems must deal with the various, machine-dependent ways that these exceptions are presented to the client. In general, these exceptions require that the computation that is responsible for the exception event be diverted from its normal path of execution, and vectored to some exception handling procedure.

When an exception occurs in Alpha, the objects affected are cleaned up (i.e., made consistent) by executing each of the abort block handler code sections from the head of the thread, back up stream until the proper level has been reached. This allows a thread to be cleaned up from within (by the thread itself, executing with its proper attributes), and this activity can span multiple objects and nodes as the thread works its way back upstream.

The asynchronous diversion of a thread's flow of control can leave an object in an inconsistent state. The system provides its own exception handling code to perform the cleanup necessary to ensure that system data structures are left in a consistent state following an exception (and that system resources are not "lost"). In addition to the system's exception handling code, the client can also provide application-specific exception handlers to perform specialized recovery/compensation actions, that can restore the application 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. 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.

6.3.2 Optimizing for Exceptions

The robustness of Alpha is enhanced by optimizing the design and implementation for the exception 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, time constraint is satisfied, or no machine exception occurs; the exception case is that the transaction aborts, time constraint is not met, or a machine exception occurs. Performance is optimized in 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 that is used to provide the maintenance of threads as they extend across nodes. The normal case for this protocol is that the continuity of the thread is not broken by failures in nodes (or the communications network); 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 extends across, rather than using only end-to-end time-outs on the remote invocations.

Furthermore, 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 does not include a (explicit) reference to the physical location of the intended entity. The expected case for this function would be that the hint is correct, while the exception case would be that the hint was wrong. The optimization of the exception case's performance is achieved by performing a multicast-like message transmission on each remote invoke, in order to locate the (dynamically relocatable) entities on a per-invocation basis. This is done instead of using a hint about the entity's physical location to perform a point-to-point remote procedure call, with a special sub-protocol that is to be performed to locate the target entity when the hint is wrong. While hints might speed references in a static system, when drastic 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.

6.4 Adaptability

One of the reasons for choosing the form of distributed computer system upon which the Alpha operating system is to execute has to do with 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 applied) application-specific information. In particular, a wide range of application- and system-specific attributes can be associated with computations and carried along with threads as they execute within the system. This 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 change.

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 the object-oriented programming model supported by the operating system, and adaptability within the operating system itself is provided by a policy/mechanism separation approach [Hansen 70].

6.4.1 Object-Oriented Programming

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].

For reasons of adaptability, the attributes of threads (such as time constraints) are (by convention) modified in a block-structured and strictly nested within the operations of objects (e.g., a time constraint block begins and ends within the same object operation). This type of constraint provides 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.

The well-defined interfaces defined by objects not only support information hiding by separating their interface specification from their internal implementation, but they also permit all system services, including devices (as well as special hardware augmentations), to be presented to the programmer as objects. This feature allows the system to be modified by substituting, interposing, or overlaying modified objects, without affecting existing code. Furthermore, Alpha supports the simple and effective downward migration of functionality by allowing system functionality to be developed as client objects (for reasons of convenience) and then migrated into the kernel (for reasons of performance).

6.4.2 Policy/Mechanism Separation

The concept of policy/mechanism separation has been shown to be valuable in the design of modular operating system facilities [Hansen 71, Levin 75, Habermann 76]. Briefly, a policy is defined as a specification of the manner in which a set of resources are managed, and a mechanism is defined as the means by which policies are implemented [Hansen 70]. Policy/mechanism separation is a structuring methodology that involves the segregation of entities that dictate resource management strategies from entities that implement the low-level tactics of resource management.

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. If the mechanisms are in fact pure (i.e., devoid of policy decisions) and complete, then it is possible to use them in implementing a wide range of system- and application-level facilities, and indeed entirely different operating systems. 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.

The Alpha operating system supports policy/mechanism separation which allows the easy addition or modification of resource management policies—most of the major subsystems consist primarily of frameworks into which a wide range of differing policies can be inserted. Furthermore, the mechanisms that provide the Alpha programming model allow specialized, application-specific policies to be developed, either at system-build time or at run-time. The system does not enforce a particular policy on the user, but rather provides mechanisms that allow a wide range of policies to be implemented (e.g., in the area of concurrency control and exception handling).

An example of where the effects of policy/mechanism separation within the system are visible at the programming interface level can be seen in the kernel's mechanisms for managing object creation. The kernel mechanism that permits the creation of new objects does not include a policy for the placement of the newly created object. The kernel creates a new object local to the node at which the creation operation was invoked, instead of choosing a "desirable" location for the new object (e.g., on the least loaded node, on the node with the greatest amount of common data, or on the node nearest the needed I/O devices). In this way, the kernel's clients are free to apply the appropriate, possibly application-specific, policy to the initial placement of objects. This applies as well to the creation of replicated objects and the initial placement of the individual replicas. Furthermore, the kernel's native, flat object space allows arbitrary constraints on object relationships to be applied at the higher levels, to achieve the desired form of structure among objects, without suffering from potentially conflicting, system-imposed structures.

7 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 in Alpha. Doug continues to provide support and guidance for this research effort an enhanced Release 2 of the Alpha operating system is being developed at Kendall Square Research Corporation.

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. Ray Clark worked on defining and implementing the basic system abstractions and is now performing his thesis research in the area of real-time transactions within the Alpha context.

Sam Shipman and worked on various subsystems and support tools, and worked on 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.

Other project members that contributed to the project are Jeff Trull, Chuck Kollar, Bruce Taylor, Don Lindsay, and Dan Reiner.

Thanks is 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.

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

[Accetta 86] Accetta, M. J., Baron, R. X., Golub, D., Rashid, R. F., Tevanian, A. and Young, M.
A New Kernel Foundation for UNIX Development.
In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, pages 35-41, June, 1986.

[Avizienis 78] Avizienis, A.
Fault-Tolerance: The Survival Attribute of Digital Systems.
Proceedings of the IEEE 66(10):1109-1125, October, 1978.

[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.

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

[Boehm 81] Boehm, B. W.
Advances in Computer Science and Technology: Software Engineering Economics.
Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[Cheriton 84] Cheriton, D. R.
The V Kernel: A Software Base for Distributed Systems.
IEEE Software 1(2):19-43, January, 1984.

[Clark 83] Clark, R. K. and Shipman, S. E.
The Archons Testbed—A Requirements Study.
Archons Project Technical Report #83051, Department of Computer Science, Carnegie-Mellon University, May, 1983.

[Clark 88] 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.

[DRC 86] Dynamics Research Corporation
Distributed Systems Technology Assessment for SDI.
Technical Report E-12256U, Electronic Systems Division, USAF Systems Command, September, 1986.

[Farber 72] Farber, D. J. and Larson, K. C.
The System Architecture of the Distributed Computer System—The Communications System.
In *Proceedings, Symposium on Computer-Communications Networks and Teletraffic*, pages 21-27. Polytechnic Institute of Brooklyn, April, 1972.

[Fitzgerald 85] Fitzgerald, R. and Rashid, R. F.
The Integration of Virtual Memory Management and Interprocess Communication in Accent.
In *Proceedings, Tenth Symposium on Operating System Principles,* pages 13-14. ACM, November, 1985.

[Franta 81] Franta, W. R., Jensen, E. D., Kain, R. Y. and Marshall G. D.
Real-Time Distributed Computer Systems.
Advances in Computers 20:39-82, 1981

[GD 80] General Dynamics
Computer Program Product Specification for the System Function Processor Operational Flight Program for the F-16 Multination Staged Improvement Program, Block 30.
Technical Report #CPCI 7175-1A00, General Dynamics Corporation, December, 1980.

[Gifford 79] Gifford, D. K.
Weighted Voting for Replicated Data.
In *Proceedings, Seventh Symposium on Operating Systems Principles,* pages 150-162. ACM, December, 1979.

[Glass 80] Glass, R. L.
Real-Time: The “Lost World” of Software Debugging and Testing.
Communications of the ACM 23(5):264-271, May, 1980.

[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.

[Hansen 71] Brinch Hansen, P.
RC4000 Software Multiprogramming System.
A/C Regnecentralen, Copenhagen, 1971.

[Herlihy 86] Herlihy, M. P.
Using Type Information to Enhance the Availability of Partitioned Data.
Technical Report CMU-CS-85-119, Department of Computer Science, Carnegie-Mellon University, April, 1985.

[Henize 84] Henize, J. A.
Understanding Real-Time UNIX.
MASSCOMP Technical Report, January, 1986.

[Jensen 75] Jensen, E. D.
Time-Value Functions for BMD Radar Scheduling.
Technical Report, Honeywell System and Research Center, June, 1975.

[Jensen 76a] Jensen, E. D.
The Implications of Physical Dispersal on Operating Systems.
Workshop on Distributed Processing, Brown University, Providence, Rhode Island, August, 1976.

[Jensen 76b] 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 78a] Jensen, E. D.
The Honeywell Experimental Distributed Processor—An Overview.
Computer 11(1):137-147, January, 1978.

[Jensen 78b] Jensen, E. D., Marshall, G. D., White, J. A. and Helmbrecht, W. F.
The Impact of Wideband Multiplex Concepts on Microprocessor-Based Avionic System Architectures.
Technical Report AFAL-TR-78-4, Honeywell Systems & Research Center, February, 1978.

[Jensen 84] Jensen, E. D. and Pleszkoch, N.
ArchOS: A Physically Dispersed Operating System.
In *Distributed Processing Technical Committee Newsletter*. IEEE, June 1984.

[Jones 79] Jones, A., Chansler, R., Durham, I., Schwans, K. and Vegdahl, S.
StarOS, a Multiprocessor Operating System for the Support of Task Forc-

es.
In *Proceedings, Seventh Symposium on Operating System Principles*, pages 117-127. ACM, December, 1979.

[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.

[Lechovski 86] Lechovski, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review 14(1):44-53, May 1986.

[Lehman 85] Lehman, M. M. and Belady, L. A.
Program Evolution: Processes of Software Change.
Academic Press, London, 1985.

[Leinbaugh 80] Leinbaugh, D. W.
Guaranteed Response Times in a Hard-Real-Time Environment.
IEEE Transactions on Software Engineering SE-6(1):85-91, January, 1980.

[Levin 75] Levin, R., Cohen, E., Corwin, W., Pollock, F., and Wulf, W.
Policy/Mechanism Separation in Hydra.
In *Proceedings, Fifth Symposium on Operating Systems Principles,* pages 132-140. ACM, November, 1975.

[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.

[Liu 73] Liu, C. L. and Leyland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
Journal of the ACM 20(1):46-61, 1973.

[Locke 86] Locke, C. D.
Best-Effort Decision Making for Real-Time Scheduling.
Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, May, 1986.

[Maynard 88] Maynard, D. P., Clark, R. K., Kegley, R. B., Keleher, 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.

[McQuillan 80] McQuillan, J. M., Richer, I. and Rosen, E. C.
The New Routing Algorithm for the ARPANET.
IEEE Transactions on Communications COM-28(5):711-719, May, 1980

[Metcalf 72] Metcalf, R. M.
Strategies for Interprocess Communication in a Distributed Computing System.
In *Proceedings, Symposium on Computer-Communications Network and Teletraffic,* pages 519-526. Polytechnic Institute of Brooklyn, April, 1972.

[Mockapetris 77] Mockapetris, P. V., Lyle, M. and Farber, D. J.
On the Design of a Local Network Interface.
In *Proceedings, Information Processing 77.* IFIP, 1977.

[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 87] Northcutt, J. D.
Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel.
Academic Press, Boston, 1987.

[Northcutt 88a] Northcutt, J. D.
The Alpha Operating System: Programming Model
Archons Project Technical Report #88021, Department of Computer Science, Carnegie-Mellon University, February, 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.

[Ousterhout 80] Ousterhout, J. K., Scelza, D. A. and Sindhu, P. S.
Medusa: An Experiment in Distributed Operating System Structure (Summary).
Communications of the ACM 23(2):92-105, February, 1980.

[Parnas 77] Parnas, D. L.
Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems.
Technical Report 8047, Naval Research Laboratory, December, 1977.

[Quirk 85] Quirk, A. B.
Verification and Validation of Real-Time Software.
Springer-Verlag, Berlin, 1985.

[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.

[Ready 86] Ready, J. F.
VRTX: A Real-Time Operating System for Embedded Microprocessor Applications.
IEEE Micro 6(4):8-17, August, 1986.

[Savitzky 85] Savitzky, S. R.
Real-Time Microprocessor Systems.
Van Nostrand Reinhold, New York, 1985.

[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.

[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.

[Stankovik 88] Stankovik, J. A.
Real-Time Computing Systems: The Next Generation.
Technical Report COINS 88-06, University of Massachusetts, January 1988.

[Smith 79] Smith, R. G.
The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver.
In *Proceedings, First International Conference on Distributed Computing,* pages 185-192. IEEE, October, 1979.

[Thomas 78] Thomas, R. H., Schantz, R. E. and Forsdick, H. C.
Network Operating Systems.
Technical Report 3796, Bolt, Beranek and Newman, 1978.

[Thompson 80] Thompson, J. R., Ruspini, E. H. and Montgomery, C. A.
TAC C^3 Distributed Operating System Study.

Technical Report RADC-TR-79-360, Operating Systems, Inc. for Rome Air Development Center, January, 1980.

[Wallis 84] Wallis, D. E.
DSDB/DTN Machine Level Architecture and Principles of Operation.
Internal Technical Report IBM-2849070, May, 1984.

[Wirth 77] Wirth, N.
Towards a Discipline of Real-Time Programming.
Communications of the ACM 20(8):557-585, August, 1977.

[Wittie 79] Wittie, L. D.
A Distributed Operating System for a Reconfigurable Network Computer.
In *Proceedings, First International Conference on Distributed Computing Systems,* pages 669-677. IEEE, October, 1979.

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