Alpha/Mach Integration Study

A proposal submitted to

Rome Laboratory

Griffiss Air Force Base

Rome, NY 13441-5700

in response to

RFP F30602-93-R-0016

Purchase Request No. C-3-2233

Prepared by the

Research Institute

of the

OPEN SOFTWARE FOUNDATION, INC.

Submitted: 5 April 1993
Accepted: 24 Sept. 1993

This is an abridged version of the proposal.

OSF RI proposes to perform a set of design investigations into synthesizing the capabilities of the Alpha Operating System[Jensen90] into the Mach Operating System[Loepere92]. The results of these investigations will be incorporated into prototype versions of the Mach system, resulting in a version of the Mach operating system that has been significantly enhanced to encompass real-time, mission-critical, distributed applications as are found in military warfare applications, such as battle management, and C^3I systems.

Principal Investigator:

Dr. Ira Goldstein
OSF Research Institute
1 Cambridge Center
Cambridge, MA 02142
617-621-7342
Fax: 617-621-8696
irag@osf.org

Technical Manager:

Douglas Wells
OSF Research Institute
1 Cambridge Center
Cambridge, MA 02142
617-621-7366
Fax: 617-621-8696
dmw@osf.org

Trademark Attributions

Open Software Foundation, Inc.
Proposal to Rome Laboratory under
RFP F30602-93-R-0016
Purchase Request No. C-3-2233
"Alpha/Mach Integration Study"

Table of Contents

1. Overview

OSF RI proposes to perform a set of design investigations into synthesizing the capabilities of the Alpha Operating System[Jensen90] into the Mach Operating System[Loepere92]. The results of these investigations will be incorporated into prototype versions of the Mach system, resulting in a version of the Mach operating system that has been significantly enhanced to encompass real-time, mission-critical, distributed applications as are found in military warfare applications, such as battle management, and C^3I systems.

Successful results of this work will be the basis of proposals to modify future versions of the OSF/Mach Kernel Interface Specification[Loepere92a]. This specification, which is being developed by the RI on behalf of ARPA, is expected to be the basis for a broad spectrum of commercially available operating systems, including Paragon OSF/1, which is now being shipped with the Intel Paragon supercomputer. Inclusion of the successful results of this work into the OSF/Mach Kernel Interface Specification will result in the evolution of commercial-off-the-shelf (COTS) operating systems that will support applications that are larger, more complex, and more distributed than those traditionally served by commercially available operating systems.

OSF's basic approach to this work is to integrate selected capabilities of Alpha into the standard Mach kernel operating system. By working with the standard Mach system, we will continually maintain conformance to the Mach Kernel Interface specification and preserve the special characteristics of Mach that have made it useful to the computing community, such as symmetric multiprocessing support and hardware independent virtual memory.

We will analyze the Alpha operating system in order to determine those aspects that are most important to satisfying the requirements of real-time, mission-critical, distributed systems, such as are found in Command and Control, and Battle Management. The results will be compared to the underlying kernel mechanisms of the Mach operating system. Based on this analysis, we will design a significantly enhanced operating system. This system, which will be based on and compatible with Mach, will include those aspects of Alpha that have been determined to be necessary in the initial analysis. E. Douglas Jensen, the creator of the Alpha operating system, will be a consultant to the effort providing a clear insight into the purpose and heritage of the Alpha operating system design.

Our research program will comprise five primary technical research tasks, as follows:

- Task 1 - Investigate and Analyze Alpha and Mach. We will investigate and carefully analyze the capabilities of the Alpha and Mach kernels to determine the fundamental kernel mechanisms of each system. In addition, we will analyze the Mach Kernel Interface Specification and make recommendations for its extension or modification.
- Task 2 - Design Enhanced Operating System. We will design real-time, distributed operating system enhancements to Mach that incorporate the real-time scheduling, transnode threads, and other architectural principles of Alpha with the underlying kernel mechanisms already present in Mach. Particular attention will be paid to the problems inherent in integrating two systems with differing design objectives.
- Task 3 - Implement Enhanced Operating System. We will implement proof-of-concept versions of the new real-time, distributed operating system enhancements that were designed as part of Task 2. Within the limits of the resources available, the emphasis will be on imple-

menting at least some version of each capability within the range of capabilities identified in Task 1, as well as resolving potential problems identified during Task 2.

- Task 4 - Demonstration of the Enhanced Operating System. We will implement and conduct a demonstration of the kernel extensions designed in Task 2 and implemented in Task 3.
- Task 5 - Documentation. We will document the technical work performed, with particular attention paid to “lessons learned.”

2. Technical Understanding of the Problem

2.1. Mission Requirements

A system is real-time if proper operation requires that results be both correct and timely. Real-time systems generally involve interaction with the physical world. Traditional real-time systems require both predictable performance and priority scheduling. Requirements for timely response range from tens of microseconds (as might be required by some process control systems) to seconds (as might be required in on-line transaction processing systems).

Traditional real-time concepts and techniques have been successful in solving problems in the context for which they were intended—relatively simple, centralized subsystems that monitor and control physical processes. Typically, however, systems that provide “hard” real-time are special-purpose and often proprietary, while more general-purpose systems only support “soft” real-time.

The last several years have seen a flurry of activity in the study, design and application of real-time, distributed systems. With the arrival of “enterprise” systems in the commercial world, the small, centralized real-time subsystems have sometimes been connected to the corporate network to provide simple control and status functions. Rarely, if ever, do the real-time capabilities extend into the central network.

Much of the impetus for these developments has come from the Defense community, and there are many proposed applications of real-time, distributed systems to critical Defense problems. One particular situation has occurred in military systems with the advent of “federated” systems. Several independently developed subsystems are interconnected into one system for a common purpose, such as weapons delivery. Almost always there is no communications architecture common to the various subsystems. Thus, the interconnection protocols and solutions tend to be *ad hoc*.

In order to solve the problems associated with federated systems, many current programs are based on “integrated” systems. The subsystems have been designed from their inception to interoperate and cooperate towards a common mission purpose. Often these subsystems are physically disjoint, interconnected via physical communication media that are subject to failure and to normal communication errors. Such a real-time distributed system must be capable of delivering optimal performance under exceptional stress. Traditional real-time methods no longer work in solving the problems inherent in these distributed systems.

The Alpha operating system[Jensen90] is the result of a long line of research into the unique problems of real-time, mission-critical distributed operating systems. The application of “best-effort” algorithms, particularly to the area of scheduling[Locke86], has allowed the creation of an environment in which the application designer can effectively define the proper behavior of the system, even in the presence of faults. Inclusion of support for adaptability and fault-tolerance has led to a simple, yet powerful programming environment for real-time application designers. The current version of the Alpha operating system, however, is only available on a very limited set of systems. Furthermore, there are no standards-compliant interfaces, such as the UNIX operating system, available for use with Alpha.

With the introduction of faster and more powerful CPUs and less expensive hardware, real-time is no longer considered to be a separate application domain with specialized hardware platforms running dedicated, special-purpose operating executives. Also, many government programs now include a requirement for the use of commercially available hardware and software, known as Commercial Off the Shelf (COTS) equipment. Similarly, a number of commercial organizations recognize the need to work within open systems and reject the use of "specials."

Thus, there is a clear need and purpose in attempting to synthesize the concepts and techniques developed in a real-time, mission-critical distributed operating system, such as Alpha, into a system that is also targeted towards the commercial market, such as OSF/Mach. This version of Mach together with a corresponding version of the OSF/1 AD server would support best-effort, real-time applications within the context of a standards-compliant environment.

2.2. Alpha Kernel Operating System

The Alpha Operating System addresses a problem domain that has traditionally been served by custom, special-purpose proprietary operating systems: real-time applications that are larger, more complex, and more distributed than conventional low-level, sampled-data systems.

Traditional real-time operating systems concepts and techniques are successful in the context for which they were intended: relatively small, simple, centralized subsystems for sample data monitoring and control of primarily periodic physical processes (e.g., digital avionics flight control, signal processing). This technology does not scale up to larger, more complex, more distributed real-time systems because they are inherently more asynchronous, dynamic, and stochastic, violating the premises, such as static determinism underlying classical real-time operating systems.

The Alpha operating system, which explicitly targets these more demanding environments, initially arose as part of E. Douglas Jensen's Archons Project on new paradigms for real-time distributed computing systems at the Carnegie Mellon University Computer Science Department. The first implementation of this system was based on Sun-2 nodes. A second implementation was performed at Concurrent Computer Corporation on Concurrent Series 8000 nodes under Rome Laboratory sponsorship. Recently that implementation was enhanced, also under Rome Laboratory sponsorship, to explore aspects of B3-level security policies in multilevel secure, real-time, distributed operating systems.

The Alpha operating system is intended to provide new concepts and techniques for performing the mission-critical integration and operation of large, complex, distributed real-time systems, such as the Strategic Defense Initiative (SDI), Battle Management Command, Control, Communications, and Intelligence (BM/C^{3}I) systems. The characteristics of this application domain differ substantively not only from those of common non-real-time systems, such as networked personal workstations and throughput-oriented supercomputers, but also from the traditional real-time small, simple subsystems for low-level sampled data monitoring and control. The differences are manifest primarily in five areas of operating system requirements:

- Real-Time: meeting as many as possible of the most important aperiodic as well as periodic time constraints, despite dynamic and stochastic run-time resource contention, overloads, and faults.
- Distribution: focusing multiple physically dispersed computing nodes for the execution of large, complex, distributed computations to perform a mission.
- Survivability: preserving the mission, human life, and property in a hostile environment with limited or no repairs or downtime during missions lasting up to decades.
- Adaptability: serving a wide variety of applications, each with requirements evolving continuously over a lifetime of decades, on a dynamic technology base.
- Security: preventing unauthorized disclosure or modification of mission sensitive data with characteristics varying over time to sets of individuals or their agents who also vary over time.

2.2.1. Alpha Problem Domain

Alpha expands the domain of real-time operating systems to encompass mission-critical applications that are larger, more complex, and more distributed than conventional low-level sampled-data ones. At present, these systems have only limited automation at the system and mission levels, and depend on custom-made, special-purpose, proprietary operating systems.

The most demanding of such real-time computer systems are always found first in military warfare environments, e.g., mission support on combat platforms, battle management, C^3I. This is because the mission, and thus the computational requirements are highly dynamic and exhibit extreme uncertainties; the system resources are often limited from the outset by size, weight, and electrical power; and therei is a significant probability of being physically damaged and destroyed, with few or no repairs possible. Here insufficient performance and survivability have drastic consequences on mission success and thus the survival of human life and property. Alpha provides enabling technology for those military and civilian real-time distributed applications that have mission-critical problems for which current technology does not offer satisfactory solutions, and that justify the expense of developing, and learning to use, the requisite new technology.

2.2.2. Real-Time Computing

As the size, complexity and distribution of real-time systems grow, quantitative differences cause profound qualitative differences between the new and the traditional real-time computing concepts and techniques; the following sections explore the implications of these differences in relation to real-time requirements, distribution, survivability and adaptability.

2.2.2.1. Real-Time Computing in Large, Complex Distributed Systems

In large complex, distributed real-time computer systems, applications often exhibit extreme uncertainties at the mission, and thus system, levels; combat systems are the clearest example. The law of physics involved in distribution exacerbate this by imposing variable, unknown communication delays between concurrently executing nodes. Thus, a significant portion of the system run-time behavior becomes unavoidably asynchronous, dynamic, and stochastic. Despite this, maximal dependability of effectiveness, survivability, and safety are vital.

Historically, the means for handling these conflicting imperatives of accommodating extreme uncertainty while assuring maximal dependability have come to rely entirely of the talent and expertise of the system's human operators in aircraft cockpits, ship bridges, and plant control rooms. However, the increasing complexity and pace of system missions, and the increasing number, complexity, and distribution of system resources, require that these operators receive more support from the system itself. While some support can be provided by application software assistance (e.g., the real-time, rule-based Pilot's Associate and similar projects), much more support must be derived through innovative advances in system integration and operation at the operating system levels.

The operating system in large, complex, distributed real-time systems must perform global management of dynamic and stochastic resource dependencies, concurrency, consistency, overloads, faults, errors and failures, and do all of this in a robust, adaptable way to meet as many as possible of the application activity time constraints most important under the current mission and resource circumstances.

2.2.2.2. Traditional Real-Time

The requirement for the operating system to provide global dependable, timely system integration in the face of extremely uncertain mission and resource conditions is diametrically opposed to the traditional real-time operating system objective of exploiting or even imposing maximal certainty of behavior—not only ends, but also means—under a relatively small number of rigidly constrained (and often unrealistic) resource and mission conditions that are anticipated and tested *a priori*. Conventional real-time operating system technology is reasonably successful in the context for which it was intended: relatively small, simple, centralized subsystems for low-level monitoring of sampled data monitoring and control of primarily periodic physical processes. However, this static, deterministic model and methodology are a simple special case that does not scale up to larger, more complex, more distributed real-time systems.

2.2.2.3. Alpha's Approach to Real-Time

Alpha introduces an entirely new paradigm for real-time computing. It manages all system resources, both physical (e.g., processor cycles, physical memory, I/O bandwidth) and logical (e.g., computations, virtual memory, semaphores and locks), in an integrated fashion. Moreover, it does so according to application-specified policies for directly employing actual application time constraints, whether hard or soft, for the production of results, thus eliminating the notorious complexities of mapping time into priorities. The Alpha time constraints have orthogonal facets of *urgency* and *importance*, which are expressed in the form of a function for each result that relates the value of producing each result to the time at which it is produced; deadlines are a simple special case, a binary-value, downward step. These "time-value functions" are readily derived by the user from the physical nature of the application environment, and may evolve dynamically over the course of a mission.

2.2.3. Distributed Computing

2.2.3.1. Mission-Oriented Real-Time Systems

The computing hardware in large, complex, distributed real-time systems consists of multiple nodes or subsystems (uniprocessor, multiprocessor, parallel processors, etc.) physically dispersed (e.g., around a ship or factory); thus they are loosely coupled via I/O communications, without directly shared primary memory.

The important consequence of physical dispersal is the absence of the global system state on which classical operating system techniques rely: this is what makes a system distributed, whether its physical dispersal is across a continent or across a chip. Distribution has major effects on the structure and functionality of both the application and the operating system.

A real-time distributed system is mission-oriented, and thus its nodes are generally peers cooperatively executing an integrated set of distributed (transnode) applications, rather than each node autonomously serving users and goals.

These peer-node applications are most naturally and effectively decomposed along logical boundaries rather than the physical node boundaries imposed by the usual network paradigm. The freedom to structure an application this way is provided by true distributed programming: the application can be written and executed as though many or all of the nodes form a virtual single computer. Thus, any application can access any resource—code, data items, devices, processors—exactly as though everything were local to it (i.e., transparently and reliably with respect to physical nodes.) In addition to the improved cost-effectiveness of such structural freedom, other advantages are gained: static and dynamic reconfiguration for error and failure recovery, function expansion or contraction, and performance enhancement can also be transparent to the application.

2.2.3.2. Conventional Networks

A conventional network-based distributed operating system, such as Mach, CRONUS, or SDOS, is simply a centralized local operating system with additional standardized interfaces and protocols for certain forms of resource sharing around separate applications executing in a network of autonomous nodes. These interfaces and protocols are typically implemented as utilities at the higher layers of the operating system; this minimizes impact of distribution on local operating systems, but severely limits operating system support for distributed applications.

Such operating systems typically offer only crude remote procedure call (RPC) programming modes layered on top of a local operating system; the node location is not at all transparent, and often internode relationships are cast into a centralized client/server model of computing.

A traditional distributed operating system delegates responsibility for almost all distributed resource management, such as, correctness of distributed execution and consistency of distributed data, to some or all of the users at very high recurring development costs, and very high recurring performance penalties.

2.2.3.3. Alpha Distributed Computing

The Alpha kernel provides a new object-oriented programming model, which is well suited for writing large, real-time, distributed software. It is based on distributed threads (loci of control), which span objects and physical nodes, transparently and reliably, via operation invocation. (The Alpha Programming Model is described in more detail below.) Physical location is transparent to thread execution except when the programming may desire to see or control some actions, such as configuration management or solving certain service outages. This programming model supports coherent distributed programming, not only for the applications, but for the operating system itself.

Alpha is a layered global operating system in that it incorporates real-time distributed transnode resource management policies and facilities to accommodate the difficult realities of asynchronous, real internode concurrency of execution and data access; of variable, unknown internode communication delays that prevent global ordering or events; and of complex failure modes. Thread integrity is maintained by Alpha despite node and communication errors and failures, with at-most-once operating invocation protocols and orphan detection and elimination protocols.

2.2.3.4. Survivability

Survivability in a larger, more complex, more distributed system is generally more important because of the supervisory nature of its applications. It is also relatively unfamiliar territory for real-time operating systems, which usually deal only with local subsystem rather than distributed system and mission level problems. Survivability at these higher levels calls for inclusion in the operating system of technology for distributed computational integrity, graceful degradation, and for continued availability of situation-specific services and data, despite complex (partial, bursty in both time and space) failures.

Maintaining integrity of distributed computations is particularly interesting and important; it means more than per-node processing integrity and internode communication integrity, which generally suffice in a network context. In addition, data that is partitioned and replicated across multiple nodes must adhere to application-specific consistency constraints, while a process that is distributed across multiple nodes must adhere to application-specific correctness constraints. Today, most technology for distributed computational integrity, as well as that for availability and graceful degradation (e.g., from the distributed database world), is developed without consideration for real-time requirements.

2.2.3.5. Adaptability

Real-time application environments demand maximum computing capability for the allowable size, weight, and power; this argues for special-purpose operating systems. But there are many widely divergent real-time applications, and the very high costs of developing special-purpose operating systems argues for the reusability of operating systems. Adaptability is more important for operating system reusability than generality, because adaptability allows maximal exploitation of application-specific requirements in reaching special-purpose/general-purpose operating system cost/performance trade-offs.

Adaptability presents special challenges in large, complex, distributed real-time systems because the computing requirements typically are ill-defined initially and continue to evolve, not only during the design phase, but across the entire system lifetime, which is often decades.

Alpha strives for a high degree of adaptability by strictly adhering to the philosophy of *policy/mechanism* separation. This means that Alpha has a kernel of primitive mechanisms from which everything else is constructed according to a wide range of possible application-specific policies to meet particular functionality, performance and cost objectives. Alpha kernel mechanisms are intended to provide the lowest meaningful (not simply the lowest possible) level of functionality for an application: anything less would require the application to do it, resulting in recurring, inconsistent, inefficient efforts, and thus a much less cost-effective system; anything more would limit policy flexibility and thus limit system cost/performance trade-offs.

2.2.4. Alpha Programming Model

The Alpha operating system kernel provides a new programming model that is well suited to writing distributed real-time software[Northcutt87]. It presents its clients with a coherent computer system that is composed of an indeterminate number of physical nodes in a reliable, network-transparent fashion. Its principal abstractions are objects, operation invocations, and threads. These are augmented by other abstractions, including capabilities. We describe these below.

2.2.4.1. Objects

Alpha objects are passive abstract data types (code plus data) on which there may be any number of concurrently executing activities (threads). Each instance of an Alpha client object has a private address space. An instance of an object exists entirely on a single node. Objects can be dynamically migrated among nodes; initial object placement is specified by the user. Objects can be transparently replicated, with members of the replicated set residing on different nodes. Alpha objects are intended to normally be of moderate number and size, e.g., 100 to 10,000 lines of code. This size is dictated by the cost of object invocation. The kernel defines a suite of standard operations that are inherited by all client objects, and these standard operations can be overloaded (redefined).

Objects are named by capabilities, which are protected by the kernel and not directly accessible by applications. Capabilities provide a network-location-transparent space of unique names.

2.2.4.2. Operation Invocation

The invocation of an operation on an object is the vehicle for all interactions in the system, including operating system calls. Threads move from object to object via invocations. Operation invocation has synchronous request/reply semantics, similar to RPC; operations are block-structured. It is straight-forward to construct asynchronous semantics on the native mechanisms, if desired. Parameters are logically passed by value.

2.2.4.3. Threads

An Alpha thread is an activity that moves among objects via operation invocation. It is a distributed computation that transparently and reliably spans nodes. A thread carries parameters and other attributes related to the nature, state, and service requirements of the computation it represents. Its attributes may be modified and accumulated, typically in a nested fashion, as a thread executes operations within various objects. Unlike with RPC or message passing in other systems, Alpha utilizes these attributes to help perform resource management on a system-wide, decentralized basis.

Alpha threads are the unit of schedulability and are fully preemptible, even those executing within the kernel. Thus, when the scheduling subsystem detects that there is a ready thread that is more important the one currently running, the current thread can be suspended and the more important thread can be executed. The preemption costs and expected completion time of the less important thread are taken into account when making this decision. In addition, Alpha's best-effort scheduling algorithms explicitly deal with the various kinds of resource dependencies among threads, and, if appropriate, they roll back a less important thread that is blocking a more important one[Clark90]. The fully preemptible and multithreaded design of the kernel facilitates real-time behavior and allows symmetric multiprocessing within the kernel.

2.3. Mach Microkernel

Originally designed as a better way to accommodate general purpose shared-memory multiprocessors, the Mach operating system has evolved into a microkernel foundation (as of Mach 3.0), which offers a hardware-independent system programming interface (SPI) within a modular system architecture. This environment provides an common foundation for constructing new system services and configuring systems for specialized applications and new hardware architectures. Mach incorporates in one system a number of facilities that allow the efficient implementation of system functions outside the operating system kernel. This structure allows a separation of the low-level, key kernel functions that are necessary to real-time responsiveness from those higher level functions that are often specially adapted to the application environment.

2.3.1. Mach Problem Domain

Mach was conceived as a message passing system that would supply compatibility with existing operating systems. Such a system was intended to be a better solution by providing system extensibility, protection, and network transparency. It was intended to provide a hardware-independent set of basic facilities that would permit a wide variety of operating systems to be efficiently implemented on uniprocessor and multiprocessor platforms.

The first use of Mach was as the foundation of time-sharing systems running the UNIX operating system. Mach 2.5 was the basis for commercial systems from NeXT, Encore, OSF and others. The majority of these systems are used as desktop workstations.

Over the past several years, Mach has become a popular operating system base for the research community. Various groups are using Mach as a vehicle for a wide variety of research efforts, including real-time, fault-tolerance, gigabit network node processing. The USENIX Association even sponsors a separate series of workshops on the Mach operating system. Some of these research efforts, which are particularly relevant to this effort, are described below.

Being similar in structure to real-time message passing executives, the microkernel version of Mach contains no higher level facilities to interfere with real-time performance. The Advanced Real-Time Technology (ART) group at Carnegie Mellon University has developed real-time scheduling and resource management enhancements for Mach[Tokuda90]. Based on a periodic, rate-monotonic scheduling paradigm, some of this work in being incorporated into the research version of Mach 3 that is available from CMU.

OSF also recently completed its first real-time project for Mach, producing a version of Mach capable of supporting real-time applications characterized by soft deadlines with moderate latencies. This microkernel is packaged with a version of the OSF/1 MK server, allowing applications to service real-time needs while also having access to the full capabilities of the OSF/1 operating system.

The Center for High Performance Computing is developing Mach/RT, which is a real-time version of the Mach 3 microkernel. Focusing on separation of scheduling policy from scheduling mechanism, support for more predictable kernel preemption, and support for resource reservation, Mach/RT is targeted at parallel and distributed applications with hard real-time requirements.

CMU and OSF have enhanced Mach to support massively parallel processor machine architectures as well as clusters of computers interconnected by high-speed communication channels. Based on an enhanced IPC mechanism for NORMA class multicomputers[Barrera91], this version of Mach is the microkernel foundation of OSF/1 AD[Zajcew93], which is the basis of Paragon OSF/1, the operating system for Intel's MPP supercomputer.

2.3.2. Mach System Architecture

There are several key characteristics of the Mach microkernel architecture. These fundamental concepts have allowed Mach to be the base for a wide variety of operating system environments:

- Portability. Most of the code in Mach is hardware independent. Those portions that are machine dependent are separated into distinct modules in order to simplify implementation on new hardware platforms.

- Network Accessibility. Mach IPC is network transparent. Thus, operating system servers need not reside on the same node as the clients that they are serving. Similarly, IPC receive "port rights," which logically distinguish the target of a message, can be moved from one task to another, even across node boundaries.
- Extensibility. A client task can obtain services from one server or many. This allows multiple operating system environments, such as UNIX and MS-DOS, to exist and support clients simultaneously.

2.3.3. Mach Programming Model

Mach provides a flexible environment for executing system and user programs. Services are provided via a few primary kernel abstractions:

2.3.3.1. Thread

A thread is the basic computational entity in Mach. A thread belongs to one and only one task. It is thus restricted to a single node. A thread is a light-weight entity with a minimum of state. The only actions that a thread can take directly are to execute instructions that manipulate its computation state, read and write within its memory space, and send and receive message. (A common optimization is to transform local message transmission into supervisor traps.)

2.3.3.2. Task

A task can be viewed as a container that holds a set of threads. It contains default values to be applied to those threads. Most importantly, it also contains those elements that its threads need to execute, namely, a port name space and a virtual address space. Tasks are restricted to a single node. (There is on-going research to allow a task to migrate between nodes[Miloj93], but even then a node must reside entirely upon a single node at any particular moment.)

2.3.3.3. Port

A port is a unidirectional communication channel between a single receiver and (potentially) multiple senders. There are certain rights associated with a port: a "receive" right, which identifies the receiver, and "send" rights, which allow the transmission of messages over the port. A port that names a service provided by a task has that task as the port's receiver; this receivership can change, if desired, by transmitting the receive right to the new server. A port right name is directly implemented by the kernel and is not directly accessible to applications.

2.3.3.4. Message

A message is a collection of data, (out-of-line) memory regions, and port rights passed between two entities via ports. A message is not a manipulatable system object in its own right, but is significant because a queued message can hold state, including port rights, between the time that a message is sent and the time that it is received.

2.3.3.5. Memory Object

An abstract memory object represents the non-resident use of the memory ranges backed by the memory object. This memory object is actually "implemented" by a memory manager task, which responds to requests to manipulate the memory object. The memory object is manipulated by sending appropriate messages over the port that “names” the memory object.

2.3.3.6. Other Abstractions

There are a number of other abstractions including: ledgers, which provide resource management; processors, which are physical hardware; hosts, which represent computers; nodes, which represent individual nodes within multicomputers; devices, such as clocks, disks, and communication channels; and events, which support a restricted form of synchronization with hardware devices.

3. Method of Approach

3.1. Key Concepts

We believe that there are two fundamental points that will make this program successful: the use of OSF/Mach as a base, and the use of distributed threads as a fundamental abstraction.

3.1.1. Use of OSF/Mach as a Base

We propose to start with OSF/Mach and add Alpha-like enhancements to that system. We intend to regularly integrate the facilities developed under the effort into the standard RI OSF/Mach source base. There are several benefits to this approach:

- The standard RI OSF/Mach base system is used daily to support development. This same base is regularly subjected to a QA cycle in which both stress and conformance tests are performed. This guarantees the integrity of the basic capabilities of Mach, including symmetric multiprocessing, hardware independent virtual memory, external paging, capabilities, shared/partitioned memory, and interprocess communication. Because both OSF and its collaborators run a variety of operating system servers on this base, the microkernel principles are maintained.
- The enhancements made to the mainline OSF/Mach base system will be available to support the Alpha-derived enhancements. Several projects that will start within the next few months typify the facilities that would otherwise be unavailable.
    - First, the *x*-kernel protocol framework will be implemented in the Mach kernel in support of a new implementation of NORMA IPC. This facility will significantly simplify the development of Alpha communication protocols, including TMAR and object invocation.
    - Second, the new implementation of NORMA IPC may be an effective replacement for the communication layer underlying Alpha RPC.
    - Finally, OSF Engineering will be adding virtual memory enhancements that were added to OSF/1 after CMU began work on Mach 3.
- There are also a few unscheduled tasks that might prove useful. For example, we are considering a subset of ISIS technology to provide reliable node failure detection. There is a similar mechanism within the Alpha TMAR facility. More than likely one mechanism would provide a solution in both environments.
- The Alpha-derived enhancements will receive a broader review within the Mach group at OSF RI. Because these changes will be subject to the design and code reviews that are com-

patible with those being used for the trusted version of OSF/Mach, more Mach developers will understand them; there will be more people with better ideas for how to solve the conceptual problems that will arise.

- Because the Alpha-derived enhancements will be included in the regular OSF RI distributions, (partial) Alpha capabilities will be available to a broad spectrum of users. Because Alpha has always run on unique and/or expensive hardware platforms, it has never before had this kind of exposure.

3.1.2. Use of Distributed Threads as a Fundamental Abstraction

We believe that the proper choice of a Mach analog for an Alpha distributed thread is key to the successful transfer of Alpha facilities into Mach. The concept that there are individual computations, each accruing value, each capable of being individually scheduled is key to the concept of best-effort scheduling, whether that scheduling involves just processor cycles or includes complex resources such as transactions and communication channels. If individual segments of the computation are not associated with the larger computation, the best-effort scheduler will make incorrect decisions. To the extent that a “computation” in an enhanced Mach appears like an Alpha thread, the existing Alpha best-effort scheduler can be easily converted to the Mach environment.

We presently believe that the best way to implement an Alpha-like object is to use a Mach task. We also believe that the best way to implement an Alpha-like “computation” in a Mach system is to use Mach threads within several Mach tasks. To emulate Alpha object invocation, we propose using standard Mach IPC with extensions to provide Alpha-like marshalling of parameters and results. The already proposed “implicit data” extensions to Mach IPC would be used to propagate the scheduling parameters, including urgency and importance, that are required by the Alpha best-effort scheduler.

This would result in Alpha-like “computations” in Mach, without necessitating drastic changes to Mach. Emulated object invocation would result in the transmission of parameters, both capabilities and standard parameters, to a standard Mach thread in the appropriate Mach task. Because the scheduling information would be passed as part of the implicit data in a Mach IPC message, the best-effort scheduler would see the execution of the individual thread components as part of one larger computation. Using simple extensions to provide the “illusion” of transtask and transnode threads, the best-effort scheduler could use the standard Mach scheduling interfaces, which are being developed now by the Center for High Performance Computing, the scheduler could manage distributed computations. This would be as true for the DASA scheduler as it is for the Locke-derived best-effort scheduler in the current Alpha system. We believe that this would also be true for the multiprocessor best-effort scheduler now being investigated.

3.2. Focus of the Research

We propose to focus particularly on the issues of defining enhancements to Mach that will allow us to take particular advantage of three lines of research that are incorporated into the Alpha operating systems: distributed computation, best-effort scheduling methodologies, and system-provided fault-tolerance support. Note that best-effort scheduling is not limited to processor scheduling[Clark90]; we propose to work with other groups who are investigating the use of best-effort algorithms on other system resources, such as the Adaptive Fault Tolerance (AFT) and System Resource Management (SRM) programs that are sponsored by Rome Laboratory.

There are several key concepts in Alpha that directly support the best-effort approach. The common element among them is that useful elements of work are divided into activities, which can be managed and controlled by system schedulers. To the extent possible, one activity is associated with exactly one Alpha thread. One Alpha thread can perform multiple activities, either serially or concurrently if nested properly. Even the fault-tolerance support provided by Alpha is reflected in the concept of activities. The relevant key concepts included in the Alpha operating system are:

- Distributed Threads
- Exceptions
- Thread Maintenance and Repair (TMAR)
- Object Invocation
- Scheduling Mechanism and Policies

There are a small number of additional concepts included in the Alpha Operating system that we believe are interesting, but not crucial to investigate. These include:

- Kernel Objects
- Real-Time Communication

In addition, we will be developing and modifying test programs and exploratory demonstration program to investigate the use and suitability of these Alpha concepts within the context of Mach.

We will incorporate aspects of these key real-time concepts into Mach in an incremental fashion. Each set of enhancements will be integrated into an specification-compliant version of Mach and offered as experimental facilities in the standard RI releases of OSF/Mach.

3.3. Technical Approach

We propose to perform the work in three stages, corresponding to several of the tasks in the RFP Statement of Work. In the first stage we will identify, analyze and study those aspects of Alpha that are most critical to the support of real-time, distributed mission-critical applications. We will also identify, analyze and study those aspects of Mach that are critical to its use as a microkernel within a modular system architecture. We will compare the basic mechanisms of the two kernel operating systems with the intent of later designing a system that integrates the facilities of Alpha and Mach into one cohesive system, rather than just add Alpha mechanisms into Mach. This additional analysis will allow us to avoid significantly increasing the size and complexity of the Mach kernel with redundant mechanisms.

The second phase is a design effort. Because we will have neither the resources nor the time to implement a completely integrated Alpha/Mach kernel, we will focus our design efforts on identifying those aspects that are crucial to providing the key Alpha concepts within Mach. We will attempt to design a spectrum of solutions, ranging from user-mode emulation of Alpha concepts to a fully integrated Alpha/Mach kernel. For those areas where there are fundamental conflicts between the Alpha and Mach concepts, we will attempt to design intermediate level mechanisms that will allow us to experiment with various solutions. We will then incorporate the results of those experiments in the final report.

The third phase is the actual implementation of the design from phase two. Although we will have created a complete design in phase two, we expect that there will be significant “lessons learned” at intermediate points. At regular intervals, we will reexamine the design from phase two in order to apply the intermediate lessons learned. We believe, however, that there is significant value is examining almost all of the key mechanisms identified in phase one. That is, we expect that in the face of restricted resources, it will be better to spend a small amount of time looking at every key concept, rather than spending a larger amount of time looking at a reduced number of key concepts. The reason for this expectation is that it is the Alpha environment in its entirety that is useful to mission-critical applications. Thus, to provide a Mach implementation of transnode threads without providing the capability to recover from node failures would not be useful to the target applications. So by the end of this effort, we intend to have spent some significant amount of time investigating each of the key concepts in Alpha.

In the descriptions of key Alpha concepts below, we discuss some of the issues that will need to be resolved during the proposed effort. Also, we describe a subset of the range of potential solutions:

3.4. Distributed Threads

The concept of distributed threads as provided in the Alpha operating system has proven to be a useful mechanism for supporting supervisory real-time applications. It is the one, single concept that enables the use of best-effort scheduling with Alpha. A thread invokes one function from another according to the requirements of the computation, crossing both object boundaries and node boundaries as necessary. Thread information, such as scheduling parameters and subject identification, is automatically passed along with the object invocation parameters. Thus, the relevant scheduling parameters are always available to the local node scheduler.

There is no analog to distributed threads in the current Mach operating system. Instead, threads are constrained to the address space of the encompassing task, each of which is confined to reside on a single node. Communication of information between tasks, whether on the same node or on different nodes, is effected via Mach IPC.

Another benefit of the Alpha model is that there is less need for a highly synchronized clock as there is in many distributed real-time systems. Clock and timing information that is relevant to a particular thread is included as part of the scheduling information when a thread moves from one node to another.

3.4.1. Level One Implementation

The simplest, most straight-forward way to provide an Alpha-like distributed threads mechanism within a Mach-defined system would be to simulate Alpha threads by using Mach IPC messages to pass object invocation parameters and results from a thread in one task to a corresponding thread in another task. In fact, the Mach system already supports such an RPC mechanism, which is normally used to communicate between threads in a client-server relationship. The element that is present in Alpha and missing in Mach is the thread information that can be delivered to the scheduler on the target node.

While this information could be passed along from Mach thread to Mach thread as extra parameters (and results) in the RPC calls, this would require every procedure call to be aware of the scheduling policy, an awareness that would clutter the design of every module in the application system. In addition, some Alpha scheduling parameters are actually resource constraints, typically limiting the resource usage of the invoked function (and all subsequently invoked functions) in order to guarantee timeliness characteristics. Allowing each such invoked function to modify the resource limits is contrary to the effective enforcement of such limits.

Thus, it would be useful to allow time-constraint information to be known by the Mach kernel and automatically passed along with the body data in Mach IPC messages. The RI is currently implementing a proposed extension[Loepere92a] to supply such "implicit data" with each Mach IPC message. In fact, one of the implicit data items that was contemplated during the development of the proposal was time-constraint information.

Just providing the raw data for time-constraints with each message does not provide a complete solution, however. Not all Mach IPC messages represent Alpha invocations: some messages are asynchronous; some messages are used to effect control in support of other threads. There must be a clear transfer of the "activity" via the Mach IPC message in order to simulate the transfer of the "activity" that occurs in an Alpha object invocation.

Thus, an interim goal towards providing Alpha-like distributed threads in a Mach-based system would be to provide a system that uses time-constraint information passed via implicit data to simulate Alpha scheduling. Components of this system would include:

- completion of the implementation of the implicit data mechanism, including time-constraint information.
- definition and implementation of a model to provide an Alpha-like thread scheduling environment. (This might include identifying certain Mach IPC messages as carrying time-constraints, i.e., being analogs to Alpha invocations.)
- definition and implementation of the Mach kernel facilities required to communicate the time-constraint information. This would include interfaces to accept the information from threads, pass the information as implicit data in particular Mach IPC messages, and accept the time-constraint information from received Mach IPC messages.
- definition and implementation of the Mach kernel facilities to communicate the time-constraint information to a local scheduler (such as the Alpha-like Best-Effort scheduler) proposed in this same set of activities.)

An additional activity to be completed in Level One would be to improve the current implementation of Mach IPC. The goals of this modification would be twofold:

- improving the real-time characteristics by reducing the execution time required. This would include general speed-ups as well as providing appropriate trade-offs for execution time versus supplied functionality.
- improving the preemptability and latency characteristics of the Mach IPC system.

3.4.2. Level Two Implementation

The Thread support package described in Level One is limited to a predefined number of threads operating in one task/object simultaneously. Level Two would eliminate this restriction:

- define and implement a scheduler activation[Anderson90] implementation for OSF/Mach
- define and implement modifications to the threads package to take advantage of scheduler activations.

3.4.3. Level Three Implementation

Although the mechanisms for management of time-constraint information provided in Levels One and Two are useful for real-time applications, they do not provide the smooth, uniform interface provided by the Alpha operating system. Thus, an activity for Level Three would be to enhance the mechanisms provided in Level One to provide an Alpha-like thread abstraction. These enhancements together with the Object Invocation Facility, which is described in this same set of activities, would allow emulation of an Alpha environment. Components of this activity might include:

- definition and implementation of enhancements to the Mach IPC system to include authorization and identification information sufficient to allow simulation of Alpha object invocation. (Note that both Mach and Alpha have ongoing projects to investigate B3-level security. Resolution of authorization and identification issues might not be consistent with the existing models. This issue will require further investigation.)
- definition and implementation of a Mach interface that would use information provided by the Object Invocation Facility to pass time-constraint information in Alpha-like object invocations implemented via Mach IPC messages.

3.5. Exception Mechanism

The Mach exception model provides simple but powerful primitives for exception handling in a system in which exceptions are the exception (i.e., in which exception processing is considered a relatively rare event, not worth optimizing). This model is used for handing faults in both the program, such as protection violations, and the environment, such as processor failure.

The Alpha exception model supports distributed exception handling, across segments of a thread that might exist on different nodes in the distributed system (using the TMAR protocol). Synchronous and asynchronous exceptions are handled uniformly. The Alpha exception handling system maintains the information about nested exception handling scopes in protected memory, to guard against corruption by a malfunctioning program.

An Alpha thread always handles its own exceptions, preserving the correspondence between the Alpha thread and the computation it is performing. (In contrast, in the Mach paradigm, a different thread must exist to handle the exceptions, since the thread that encountered the exception is suspended). The kernel must also make its own exception handling provisions, since threads can be preempted while executing in the kernel, both in Alpha and Mach.

The Alpha exception model also distinguishes between two fundamentally different classes of exception handling: normal and abort. The goal in handling a normal exception is that the affected computation be able to recover from the exception and continue somehow, either where it left off, or in some other exception handling code. Normal exception handling is provided in most systems, including Mach.

In contrast, the Alpha notion of an abort exception has a different purpose: when a computation is aborted, the object is to immediately terminate that computation (or the aborted portion of the computation) because it has been determined that the computation should not exist at all, i.e., that the computation has no legitimate claim to the resources it is using and must immediately give them up. In this case, the goal is to expeditiously terminate the aborted computation, doing any cleanup required, and execute other computations, or portions of a partially aborted computation, immediately. These differing objectives result in somewhat different strategies for exception handling in the two cases.

3.5.1. Level One Implementation

The implementation of exceptions for the Level One would be implemented as part of a library. This would allows interaction with both the scheduler and TMAR without requiring kernel modifications.

3.5.2. Level Two Implementation

Based on the design of distributed threads that is chosen for Levels Two and Three, the implementation of exceptions would need to be more integrated into the kernel. It could be implemented either within the microkernel or at user-level, or with components at both levels, depending on performance requirements. A careful study of implementation trade-offs would be done to determine which is most effective.

3.6. Thread Maintenance and Repair

Alpha distributed thread semantics require that an Alpha thread be continuous from its root (i.e., the point at which it was created) to the head (the point at which it is currently executing). If a node of the distributed system fails, the continuity of threads passing through that node is broken; this is referred to as a “thread break.”

The Thread Maintenance and Repair (TMAR) protocol monitors threads to detect breaks and repair them, by “trimming” the thread back to the point just before the break, and terminating the “orphan” thread segments left behind. This is an important component for maintaining the default “at most once” semantics of Alpha-style object invocation.

In some cases, a thread is (conceptually) deliberately broken because of some exception condition, such as a computation’s time constraint having expired before the affected thread had finished that computation. In either case, the TMAR protocol uses the exception handling mechanisms for Alpha distributed threads to effect the actual termination of orphaned thread segments at each affected node.

TMAR is an asynchronous decentralized protocol and can handle multiple successive failures as they occur. It affords a range of trade-offs between communication bandwidth used and the responsiveness of orphan detection.

3.6.1. Level One Implementation

Level One of TMAR would be implemented as part of an Alpha server. This would allow interaction with both the scheduler and the exception mechanism without requiring kernel modifications. (Note that TMAR might be implemented as a protocol graph using the x-kernel[Hutch91] framework.)

3.6.2. Level Two Implementation

Based on the design of distributed threads that is chosen for Levels Two and Three, the TMAR protocol implementation would need to be more integrated into the kernel. It could be implemented either within the microkernel or at user-level, or with components at both levels, depending on performance requirements. A careful study of implementation trade-offs would be done to determine which is most effective.

3.7. Object Invocation

Alpha's object model has proven itself to be a useful mechanism in distributed, real-time applications. Each Alpha object is isolated into an individual protection realm, which is both protected from modification or examination by code external to the object, and which is also confined from examination or modifying other objects. Within each object, multiple concurrent threads can manipulate the contents of the object subject only to the constraints imposed by the procedures comprising the object.

Object invocation is the mechanism used in Alpha to allow a procedure in one object to utilize the procedures and data in another object. It provides a method to allow one object to securely pass parameters and receive results from another object. Because it is defined in the context of Alpha's threads, invocation provides location-transparency: the method for one object invoking another object is identical, regardless of whether the target object is on the same node or a different one. Each thread retains its identity and other characteristics while executing within the context of an object.

Mach takes a different approach to solving such problems. A Mach task has many of the characteristics of an Alpha object. Each Mach task is isolated into an individual protection realm, which is both protected from modification or examination by code external to the task, and which is also confined from examining or modifying other tasks (except as explicitly permitted). Within each task, multiple concurrent threads can manipulate the contents of the task subject only to the constraints imposed by the procedures comprising the task.

A Mach thread is constrained to one task, however. Mach tasks communicate with other tasks via a traditional Inter-process Communication (IPC) mechanism. Messages can be sent from a thread in one task to a thread in another task via Mach IPC. Remote Procedure Call (RPC) is a commonly-used paradigm within Mach. A thread in one Mach task can send parameters to another task via a Mach IPC message, wait for a reply message, and extract the results from the received messages. Each task is protected from the other task except as they might choose to cooperate via the exchanged messages.

The primary difference between the two models is in the flow of control. A computation is effected in Alpha by a single thread, which moves from object to object; the operating system manages the thread by using the identification and scheduling information associated with that thread. A computation is effected in Mach by multiple cooperating threads, which are often performing other unrelated activities (unrelated except by locality of data); while each task and thread have identification and scheduling information associated with them, there is no set of identification and scheduling information associated with the basic computation. The result is that a Mach scheduler can not make decisions about advancing or retarding the progress of particular computations. It can only advance or retard the progress of particular threads, which might at various times participate in multiple computations.

Another difference is that the Alpha distributed threads model allows the operating system to determine which computations are dependent upon the operation of particular computing elements within a distributed system. Using this information, the operating system can monitor the state of the various nodes within the system. Upon detecting a change in state of one of the nodes, the operating system can notify exactly the subset of the extant computations that are affected by the node state change. Because there is no concept of an individual computation within Mach, each task must provide its own monitoring and exception recovery.

3.7.1. Level One Implementation

The simplest design would support the distributed thread simulation model by providing libraries and server support for using Mach tasks as Alpha-like objects and for using Mach IPC receive rights as Alpha capabilities.

3.7.2. Level Two Implementation

Enhance the Mach system to include mechanisms to support Alpha-like object invocation. Components of this activity would include:

- define and implement a method to provide isolation of data in the manner of Alpha stacks, but in the context of Mach tasks and threads.
- define and implement a method to identify individual computations within the context of a Mach task and/or thread.
- define and implement a method to associate system control information, including identity and scheduling parameters, with a single computation.
- define and implement an exception mechanism, similar to the Alpha Exception Mechanism, that allows reporting of changes in the state of a computation.
- define and implement a mechanism, similar to the Alpha Thread Maintenance and Repair (TMAR) facility, that provides automatic notification of node failure and recovery. This mechanism would also use the exception mechanism described above to report the node state change to the computation.

3.8. Scheduling Mechanisms and Policies:

Alpha provides a separation of the system scheduler into mechanism and policy components. The mechanism resides within the Alpha kernel. The scheduling policy used for a particular application might exist in a client object, in a kernel object, or on a dedicated purpose processor. Traditional scheduling policies, such as Earliest Deadline First, or the POSIX priority-based scheduler can easily be implemented within the Alpha context.

Locke's Best-Effort scheduler[Locke86] and Clark's DASA scheduler[Clark90] are particular examples of schedulers that implement one of Alpha's most important real-time characteristics: being able to employ arbitrary scheduling policies; and in particular, supporting scheduling policies based on the Benefit Accrual Model[Jensen93] of real-time.

These tasks transition one of Alpha's most important real-time characteristics to Mach: being able to employ arbitrary scheduling policies; supporting scheduling policies based on the Benefit Accrual Model of real-time; and providing best-effort policies. The Center for High Performance Computing has begun work on the design for separating the mechanism and policy in Mach. The first phase of this work is mostly independent of the implementation of the Alpha API.

3.8.1. Level One Implementation

Replace the current Mach scheduler with separate mechanism and policy components. Particular activities include:

- Design and implement a universal scheduling mechanism that allows the use of a wide range of policies.
- Implement the current time-sharing and real-time policies using the universal scheduling mechanism.
- Implement an Earliest Deadline First scheduling policy.

3.8.2. Level Two Implementation

Follow-on activities in the scheduling area include refinements to better support the Alpha best-effort scheduling concepts, and to support the evolving distributed threads and fault-tolerance mechanisms:

- Develop instrumentation in the scheduling facility to provide detailed performance information required to drive simulations or analytical models developed to characterize the performance of scheduling algorithms.
- Implement a best-effort scheduler based on Locke’s algorithm.

3.8.3. Level Three Implementation

Enhance the capabilities provided in Level 2. Specifically:

- Implement Clark’s Dependent Activity Scheduling Algorithm (DASA).

3.9. Kernel Objects

Alpha kernel objects are objects that are built into the Alpha kernel. Normally, kernel objects are built into the kernel in order to use privilege not otherwise available. For example, on most hardware architectures, device objects, which interact with hardware entities, such as disk controllers or network interfaces, must execute in supervisor mode in order to manipulate hardware registers. Typically, device objects must be implemented as Alpha kernel objects.

A second reason is speed of execution. While an Alpha object invocation need not be as expensive as a task switch on other operating systems, such as Mach, the mechanism is more costly than a simple trap into the kernel. Implementing certain performance critical objects as kernel objects can significantly improve the performance of a real-time system.

3.9.1. Level One Implementation

OSF is currently investigating methods for dynamically loading servers into the kernel address space. One method that may be a starting point for an OSF effort is an approach taken at the University of Utah[Lepreau93]. We expect that a similar method can be used to load either an Alpha server or client servers into the Mach kernel. This method will require little or no modifications to that server.

3.10. Real-Time Communication

While most real-time systems concentrate on processor resources, management of other resources, such as primary memory and secondary storage, is also of concern in supervisory real-time systems. In a distributed real-time system, scheduling and control of communication facilities is likely to be as important as scheduling the processors on the various nodes within the system. A real-time Mach system must incorporate facilities for controlling messages both within the nodes and while the messages are “on-the-wire” between nodes.

The eXpress Transport Protocol (XTP)[Strayer92] is a transport-level protocol explicitly designed to support real-time computation within a distributed context. This protocol supports multiple media, including fiber optics, and has been designed to allow delegation of scheduling control from the operating system. It includes rate and flow control as well as reliable multicast. In addition, it includes a 32-bit SORT field, which can be used to specify message priority or time constraints.

IPC messages are a fundamental abstraction within Mach. These messages are used to communicate both between user tasks, and between a user task and kernel-provided services. The kernel also uses IPC messages to communicate asynchronously generated information about various system resources. Clearly, the scheduling and control of transmission and delivery of IPC messages is critical to the success of a real-time application in a Mach context.

In addition, Mach IPC provides a rich set of semantics, including migration of receive rights as well as generation of other information, such as "no more senders" notifications. Many Mach applications depend on these abstractions. Thus, a version of Mach that has been enhanced to support Alpha-like computations must schedule and control these higher-level semantics in the same manner that it schedules and controls the basic messages.

Note that this activity more than others does not need to be implemented under this proposal.

3.10.1. Level One Implementation

Even a simple implementation of XTP would be useful to a Mach system with enhanced real-time facilities, with or without the Alpha-like extensions. Thus, the Level One design of the system should:

- provide a recent version of XTP. (Note that XTP might be implemented as a protocol graph using the x-kernel[Hutch91] framework.) The XTP facilities, such as rate and flow control and the SORT field, should be integrated into the standard facilities. This protocol should be available in two modes:
    - be available to user tasks as an alternative protocol within the normal suite of networking protocols.
    - also be available to be used as an alternative to TCP and other transport-level protocols within the context of Mach IPC message delivery.

3.10.2. Level Two Implementation

A Mach system controls IPC messages in several places within the system. Each message should be associated with a particular computation and scheduled in concert with that computation. The Level two design of the system should allow a user-specified scheduler to manage and control IPC-related facilities with Mach, including the following tasks:

- define and implement modifications to the Mach kernel IPC facilities that provide mechanisms to allow a scheduler to control the progress of the transmission of Mach IPC messages.
- identify other places within the Mach system that generate and/or manipulate Mach IPC messages. Define and implement interfaces to allow a scheduler to manage these messages.
- define and implement modifications to the Mach Network Message Server that provide mechanisms complementary to the Mach kernel IPC enhancements developed above.
- define and implement methods to allow a scheduler to manage and control the higher-level

semantics of Mach IPC, such as receive rights and “no more senders” notifications.

3.11. Sample Analysis

Below is a simple version of a sample analysis that we would contemplate for the Study portion of the proposed work. It contains descriptions of some possible implementations of Alpha-like distributed, transnode threads on Mach. In addition, it describes the benefits and problems associated with various methods of implementing Alpha distributed threads using the Mach microkernel. Note that we have not yet considered in depth the effect of major modifications to the programming model of either system. That issue needs to be considered as part of the proposed work.

3.11.1. Key Concepts

There are several concepts from the Alpha model that have been taken to be absolute requirements. These include:

- distributed thread concept -- the serial association of several thread components into one thread possibly distributed over multiple nodes.
- multiple concurrent access to objects -- the ability of several threads to independently and concurrently access the data of a single object.

Notes:

- All scenarios documented below describe an implementation of Alpha-like semantics on a Mach microkernel.
- Use of terms simple, complex, etc., are used in a relative sense.
- Some of the possible implementation forms require invocations to pass through an Alpha server. In all of these case, there is a problem mapping between the various Mach threads that are acting on behalf of a user and the single activity model used within the best-effort schedulers. A reasonable solution appears to be to dedicate threads in the Alpha server to service particular Mach tasks and use the newly-proposed “implicit” data IPC parameters in the RPC messages in order to meld the various threads into one activity.
- Some of the overhead of the Alpha server could be overcome by moving some of its functions into the Mach microkernel, which would result in the effect of an additional task management system.

3.11.2. Simple Emulation

The first form is a straight-forward emulation of Alpha threads. Each instance of an Alpha thread in an object would be incarnated as a Mach task with one thread. Each of the Mach tasks would have the same subject as the Alpha thread. There would be a special threads package to support the Alpha interface. There would also be a special Alpha server that would be responsible for task management, including creation, destruction, and TMAR tasks.

An invocation would consist of an RPC to the appropriate Alpha server. That server would create a new Mach task, map in the Alpha run-time support components, map in the appropriate Alpha object program and data components, and then pass the RPC to the newly created task.

Benefits:

- Simple to implement.

Problems:

- Inefficient. One invocation includes the full cost of a task create.

3.11.3. Use of Slave Tasks

This form uses a “slave” task in a manner similar to a number of operating systems that support distributed operation. The primary difference is that the Mach task is not destroyed after the first invocation returns. Instead, the Alpha server keeps track of the task. When future invocations from the same subject are received, the Alpha server maps the proper Alpha object program and data components into the address space of the slave task, and passes the RPC to the slave task.

The Alpha Server would have to keep track of currently unused Mach tasks. It would also have to make determinations about whether a particular Mach task could be used to satisfy a newly arrived RPC; a Mach task would have some attributes, such as security information, that could not be modified.

Benefits:

- Saves cost of task create on most invocations.

Problems:

- Previous invocations can damage slave tasks causing problems with unrelated invocations.
- There is now a caching policy issue: how many tasks to hold for each subject? when to release tasks from the cache? These are hard to solve for real-time systems.

3.11.4. Use of Mach Tasks as Objects

Another interesting implementation form is the use of a Mach task as an Alpha object. An Alpha server would instantiate an Alpha object by creating a Mach task and mapping the program and data components of that object into the task. An Alpha threads package would support multiple concurrent threads within the task. Each thread within the object would use a unique subject associated with the object type (or possibly the object); the Alpha subject would be maintained by the Alpha server.

An invocation would consist of an RPC to the appropriate Alpha server. The Alpha server would maintain information about the association of Alpha threads to Mach threads within the various object tasks.

Benefits:

- There is no extraneous Mach task creation on an invocation.

Problems:

- The individual thread stacks are not protected from each other. Thus, an error in one thread can damage another computation.
- There is no direct association of the Alpha subject with the executing Mach thread. Thus, the Alpha security model would need to be implemented in the Alpha server, a concept at odds with the B3-level requirements.

3.11.5. Use of Mach Threads As Objects -- with address space "forking"

An interesting modification to the concept of using Mach tasks as Alpha objects is to modify the Mach system to disassociate a Mach task from the address space. An invocation would then result in the creation of a new portion of an address space (for the thread stack) that would be associated with the common portions used for the program and data components of the object. This would also allow the Alpha subject to be associated with the Mach thread.

Benefits:

- Individual thread stacks are protected from each other. Assuming that the thread stacks are newly created on each invocation, no computation could damage future invocations (except by damaging the common data component of the object.)
- The use of the Alpha subject with the Mach thread would provide a closer association with the Alpha security model.

Problems:

- This would be a significant change to the Mach model. Determining the ramifications would involve a lot of time. For example, should IPC receive rights be associated with the address space or the threads?
- This change would invalidate the Mach security model for B3-level, but still not make the Alpha security model valid. Thus, there would need to be a future effort to define a new security model.

4. Statement of Work

In response to the Statement of Work in the RFP, we will perform the following tasks, within the constraints of time and funding.

4.1. Task 1 - Investigate and Analyze Alpha and Mach

4.1.1. Investigation and Analysis of Alpha Kernel

We will investigate and analyze the capabilities of the Alpha kernel. Specific issues to be investigated and analyzed include:

- Best Effort Scheduling within the context of the Benefit Accrual Model[Jensen93]
- Coherent Schedule Approach
- Policy/Mechanism Separation
- Object/Thread Architecture
- Fault Tolerance Support (Exception Mechanism)

4.1.2. Investigation and Analysis of Mach Microkernel

We will investigate and analyze the capabilities of the Mach microkernel. Specific issues to be investigated and analyzed include:

- Symmetric Multiprocessing
- Hardware-Independent Virtual Memory

- Use of the External Pager Mechanism
- Separation of Systems into Microkernel and Server Components
- Capabilities (port rights)
- Shared/Partitioned Memory
- Inter-Process Communication

In addition, to these specific issues, we will investigate and analyze those aspects of the Mach microkernel that most closely correspond to the specific capabilities to be investigated and analyzed in the Alpha kernel.

4.1.3. Analysis of Mach Kernel Interface Specification

We will analyze the OSF/Mach Kernel Interface Specification[Loepere92a], which is being developed under the direction of ARPA. This analysis will focus on the adequacy, completeness, and consistency of that specification for the support of real-time, distributed systems and applications. We will make recommendations regarding its extension or other modification.

4.1.4. Documentation of Task 1 Results

We will document the results of the above analyses in a technical report, which will be delivered to Rome Laboratory.

4.2. Task 2 - Design Enhanced Operating System

4.2.1. Design of Real-Time, Distributed Operating System Kernel

We will design a new integrated real-time, distributed operating system that incorporates the real-time scheduling and other architectural principles of Alpha with the underlying kernel mechanisms represented by Mach.

4.2.2. Documentation of Task 2 Results

We will document the results of that design in a System/Segment specification and a Software Design Document. Updates and/or supplements of these documents will be submitted, as appropriate, during the life of the contract. Updates will incorporate change bars or some other appropriate method to identify changes from previous versions. Final copies of these documents will be submitted at the end of the contract.

4.3. Task 3 - Implement Enhanced Operating System

4.3.1. Implement the Real-Time, Distributed Operating System Kernel

We will implement a subset of the new integrated real-time, distributed operating system that was designed as part of Task 2. The software will be implemented on a mutually acceptable computer system to be determined during the course of the effort.

The software will be designed, developed, and documented in accordance with paragraphs 4.2.8, 4.2.9, 4.6.4b,c, 5.3.2.3, 5.4.2.3, and 5.7.2.1 or DOD-STD-2167A unless otherwise determined during the course of the effort. OSF intends to request a waiver from this requirement and will suggest that the procedures being used as part of the standard OSF/RI development process for OSF/Mach should be used instead. In particular, OSF/RI is developing a version of OSF/Mach for ARPA that will conform to B3 requirements as specified in the DoD Trusted Computer System

Evaluation Criteria[TCSEC85] and will propose to use the same development procedures for this effort.

In either case, information generated and documented as the result of engineering analysis and informal testing will be included in the appropriate Software Development Files.

All software will be developed utilizing a Higher Order Language (HOL) selected from the list of HOLs designated in AFR 800-14, dated 29 Sep 86 or other language to be determined during the course of the effort. OSF intends to request a waiver from this requirement and will suggest that the same language that is used for B3 version of OSF/Mach that is being developed for ARPA.

4.4. Task 4 - Demonstration of Enhanced Operating System

We will develop a demonstration plan to implement and conduct a demonstration of the integrated real-time, distributed operating system kernel subset developed as part of Task 3.

4.5. Task 5 - Documentation

We will document all technical work accomplished and information gained during the performance of this effort. This will include all pertinent observations, the nature of problems, and positive and negative results. In addition, we will document any design criteria established, procedures followed and processes developed. Particular attention will be paid to documenting "lessons learned." Finally, we will document the details of all technical work in order to permit full understanding of the techniques and procedures used in evolving the technology or procedures that have been developed. All design, engineering, and/or process specifications developed and delivered as part of this contract will be cross-referenced to permit a full understanding of the total effort.

4.6. Task 6 - Oral Presentations

4.6.1. Rome Laboratory Distributed Systems Technology Exchange Meetings

We will attend and conduct an annual Oral Presentation at the Rome Laboratory Distributed Systems Technology Exchange Meeting to give status, results, and progress of the Alpha/Mach Integration effort.

4.6.2. Technical Progress Oral Presentations

We will conduct additional oral presentations to provide status of the technical progress made to date in the performance of this effort.

These additional oral presentations are:

- First Oral Presentation—This presentation will be held at Rome Laboratory upon completion of Task 1, in Month 6.
- Second Oral Presentation—This presentation will be held at Rome Laboratory upon completion of Task 2, in Month 10
- Final Oral Presentation—This presentation will be held at Rome Laboratory 30 months after the start of the contract.

5. Expected Results

The specific results of this effort are as follows:

- A set of proposed enhancements to the OSF/Mach Kernel Interface Specification that will support real-time, mission-critical, distributed applications on a widely available microkernel platform.
- An enhanced version of OSF/Mach, which includes experimental capabilities to support real-time, mission-critical, distributed applications.
- A sample application that demonstrates the capabilities of the enhanced version of OSF/Mach.
- A widely available vehicle for experimenting with Alpha concepts and facilities.
- Simple access to UNIX applications from within an Alpha-like environment.

6. Schedule and Technical Milestones

The following are the milestones for this effort. The schedule indicates the relative month (from Start of Contract) at which the milestone will occur. The start of the contract is October, 1993.

Milestone	Schedule	Planned Date
Start of Contract	SOC	October, 1993
Kickoff (at OSF)	SOC+1	November, 1993
Rome Laboratory Technical Exchange		Fall, 1993
Technical Report / Study / Services Complete	SOC+5	March, 1994
First Oral Presentation (At Rome Laboratory)	SOC+6	April, 1994
System / Segment Specification Complete	SOC+9	July, 1994
Software Design Document Complete	SOC+9	July, 1994
Second Oral Presentation (at Rome Laboratory)	SOC+10	August, 1994
Rome Laboratory Technical Exchange		Fall, 1994
OSF/Mach Release with Simple Distributed Threads and Objects		February, 1995
Rome Laboratory Technical Exchange		Fall, 1995
Software User's Manual Complete	SOC+27	January, 1996
Computer System Operator's Manual Complete	SOC+27	January, 1996
R&D Test and Acceptance Plan Complete	SOC+28	February, 1996
Demonstration	SOC+29	March, 1996
Final Report Complete	SOC+29	March, 1996
Final Oral Presentation (at Rome Laboratory)	SOC+29	March, 1996

7. Bibliography

Acetta, M.,Baron, R.,Bolosky, W.,Golub, D.,Rashid, R.,Tevanian, A.,Young, M., "Mach: A New Kernel Foundation for Unix Development," *USENIX Association Summer Conference Proceedings*, Atlanta, 1986.

[Anderson90] Anderson, T.E., Bershad, B.N., Laziest, E.D., Levy, H.M., Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, Technical Report 90-04-

02, Department of Computer Science and Engineering, University of Washington, 1990 (revised 1991).

Barrera, J., "A Fast Mach Network IPC Implementation," *Proceedings of the Second USENIX Mach Workshop*, Monterey, 1991.

Birman, K., "ISIS: A System for Fault-Tolerant Distributed Computing," *TR 86-744*, Cornell University, April 1986.

Branstad, M.,Tajalli, H.,Mayer, F., "Security Issues of the Trusted Mach System," *Fourth Aerospace Computer Security Applications Conference*, Orlando, 1988.

[Clark90] Clark, R.K., *Scheduling Dependent Real-Time Activities*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.

Dale, P., ed., "Industrializing Mach for Multicomputers," *OSF/1AD Project Plan*, May 15, 1991.

Forin, A.,Golub, D.,Bershad, B., "An I/O System for Mach 3.0," *CMU-CS-91-191*, Carnegie Mellon University, October 1991.

Golub, D.,Dean, R.,Forin, A.,Rashid, R., "Unix as an Application Program," *Proceedings of the Summer USENIX Conference*, Anaheim, 1990.

Hutchinson, N.,Peterson, L., "Design of the *x*-Kernel," *Proceedings of SIGCOMM '88*, 1988.

[Hutch91] Hutchinson, N.C., and Peterson, L.L., The x-Kernel: An architecture for implementing network protocols. IEEE Transactions on Software Engineering, 17(1):64-76, 1991.

[Jensen90] Jensen, E.D. and Northcutt, J.D., Alpha: An Open Operating System for Mission-Critical Real-Time Distributed Systems—An Overview, Proceedings of the 1989 Workshop on Operating Systems for Mission-Critical Computing, ACM Press, 1990.

[Jensen93] Jensen, E.D., "A Realtime Scheduling Model for Asynchronous Decentralized Computing Systems," *International Symposium on Autonomous Decentralized Systems*, Kawasaki, Japan, March, 1993.

[Lepreau93] Lepreau, J., Hibler, M., Ford, B., Law, J., and Orr, D., "In-Kernel Servers on Mach 3.0: Implementation and Performance," *Proceedings of the Third USENIX Mach Symposium,* Santa Fe, April, 1993.

[Locke86] Locke, C.D., Best-Effort Decision Making for Real-Time Scheduling, Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, 1986.

[Loepere92] Loepere, K., Mach 3 Kernel Interface, Open Software Foundation and Carnegie Mellon University, 1992.

[Loepere92a] Loepere, K., OSF Mach Draft Kernel Interfaces, Open Software Foundation and Carnegie Mellon University, 1992.

Loepere, K., *Mach 3 Kernel Principles*, OSF, 1991.

Loepere, K., ed., *Mach 3 Kernel Interface*, OSF, 1991.

Loepere, K., ed., *Server Writer's Guide*, OSF, 1991.

Loepere, K., ed., *Server Writer's Interface*, OSF, 1991.

Loepere, K., Neves, P., “Mach Microkernel Based BSD Performance,” *OSF Internal Report*, OSF, April, 1991.

[Miloj93] Milojicic, D., Zint, W., Dangel, A., and Giese, P., “Task Migration on the top of the Mach Microkernel,” *Proceedings of the Third USENIX Mach Symposium*, Santa Fe, April, 1993.

[Nakajima91] Nakajima, J., Yazaki, M., and Matsumoto, H., “Multimedia/Realtime Extensions for the Mach Operating System,” Proceedings of Usenix Multimedia Conference, June 1991.

[Northcutt87] Northcutt, J.D. *Mechanisms for Reliable, Distributed Real-Time Operating Systems: The Alpha Kernel*. Academic Press, 1987.

Open Software Foundation, *Introduction to DCE*, OSF, 1991.

Open Software Foundation, *RI Notes*, OSF, August 1991.

[Ready86] Ready, J.F., VRTX: A Real-Time Operating System for Embedded Microprocessor Applications, IEEE Micro, August 1986.

Reynolds, F., Heller, J., “Kernel Support for Network Protocol Servers,” *Proceedings of the USENIX Mach Workshop*, Monterey, 1991.

Riganati, J.,Kopetzky, D.,Minnich, R.,Fuss, D., et al., “Critical Issues for Operating Systems in Supercomputing Environments,” *Joint Workshop on Critical Issues for Operating Systems in Supercomputing Environments*, Lawrence Livermore National Laboratory and Supercomputing Research Center, September, 1991.

[Shipman92] Shipman, S., Teller, M.J., Paciorek, N., “Mach/RT Kernel Interfaces (Draft),” TR92-011, Center for High Performance Computing, Worcester Polytechnic Institute, 1992.

[Strayer92] Strayer, W.T., Dempsey, B. J., Weaver, A. C., XTP: The Xpress Transfer Protocol, ISBN 0-201-56351-7, Addison-Wesley, 1992.

[TCSEC85] —, Department of Defense Trusted Computer System Evaluation Criteria, DoD 5200.28-STD, December, 1985.

[Test92] Test, J.A., “Mach 3.0 Multimedia Real-Time Requirements”, OSF Research Institute Symposium, February 1992.

[Tokuda90] Tokuda, H., Nakajima, T., and Rao, P., “Real-Time Mach: Towards a Predictable Real-Time System,” *Proceedings of the USENIX Mach Workshop*, Burlington, November 1990.

[Travos92] Travostino, F., “Mach 3 Locking Protocol”, OSF Research Institute Technical Report, Forthcoming.

[Zajcew93] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., Netterwala, D., “An OSF/1 UNIX for Massively Parallel Multicomputers,” *Proceedings of the Winter USENIX Conference*, San Diego, CA, January, 1003.