An Architectural Overview Of The Alpha Real-Time Distributed Kernel

Raymond K. Clark
Concurrent Computer Corp.
rkc@westford.ccur.com
+1 508 392 2740

E. Douglas Jensen
Digital Equipment Corp.
jensen@helix.enet.dec.com
+1 508 493 1201

Franklin D. Reynolds
Open Software Foundation
fdr@osf.org
+1 617 621 8721

Abstract

Alpha is a non-proprietary experimental operating system kernel which extends the real-time domain to encompass distributed applications, such as for telecommunications, factory automation, and defense. Distributed real-time systems are inherently asynchronous, dynamic, and non-deterministic, and yet are nonetheless mission-critical.The increasing complexity and pace of these systems precludes the historical reliance solely on human operators for assuring system dependability under uncertainty. Traditional real-time OS technology is based on attempting to assert or impose determinism of not just the ends but also the means, for centralized low-level sampled-data monitoring and control, with an insufficiency of hardware resources. Conventional distributed OS technology is primarily based on two-party client/server hierarchies for explicit resource sharing in networks of autonomous users. These two technological paradigms are special cases which cannot be combined and scaled up cost-effectively to accommodate distributed real-time systems. Alpha's new paradigm for real-time distributed computing is founded on *best-effort* management of all resources directly with computation completion time constraints which are expressed as *benefit functions*; and multiparty, peer-structured, trans-node computations for cooperative mission management.

1. Introduction

The Alpha OS kernel is part of an multi-institutional applied research and advanced technology development project intended to expand the domain of real-time operating systems from conventional centralized, low-level sampled-data, static subsystems, to encompass distributed, dynamic, mission-level systems.

This paper begins with a summary of the distinctive characteristics of Alpha's application context: integrating constituent lower-level, centralized, real-time subsystems into one system focused on performance of a single real-time mission; and managing that system to meet (in many cases, changing) mission objectives given the current (in many cases, changing) internal and external circumstances. Real-time distributed system integration and mission management is a predominately asynchronous endeavor in which conflicts and overloads are inevitable, but most activities have hard and soft real-time constraints. These combined factors constitute the requirement for an apparent oxymoron: distributed resource management which is dynamic and non-deterministic yet nonetheless real-time.

To help resolve this conflict between needing functional and temporal dependability, and accommodating inherent uncertainty, we devised a new paradigm for real-time computing; it is founded on two concepts. The first is that real-time computations have individual and collective "benefit" (both positive and negative) to the system which are functions of their completion times; thus, maximizing *accrued* benefit can be the basis for highly cost-effective real-time acceptability criteria. The second is that in many (especially mission-level, distributed) real-time computing systems, it may be much preferable for the OS to do the best (as defined by the user) that it can under the current resource and application conditions, than for the OS to fail because these conditions violate the re-

strictive premises underlying its structure and resource management algorithms.

The paper then describes Alpha's kernel programming model, which is based on *distributed threads* that span physical nodes, carrying their real-time and other attributes to facilitate system-wide resource management. Transactional techniques are employed to maintain trans-node application-specific execution correctness and data consistency. We also synopsize the intended system configurations—where Alpha is either the only OS in the system, or supports distributed applications while interoperating with extant centralized OS's and applications (e.g., UNIX or low-level sampled-data subsystems).

Some of our architectural experiences to date with Alpha are then synopsized in the context of comparisons with related kernel work, such as Mach 3.0.

The paper concludes with a brief overview of the project history and status.

2. Distributed Computing And Its Implications On Real-Time Resource Management

Physically distributed computing arises whenever a computing system comprised of a multiplicity of processing nodes has a ratio of nodal computing performance to internodal communication performance (primarily latency but also bandwidth) which is significantly high as far as the application is concerned.

The nodal/internodal performance ratio and its significance—i.e., the degree of physical distribution—will usually be different for computations at different levels in the system. For example, a given system could have ratios which are: relatively insignificant to an application; highly significant to the "middleware" application framework, such as DCE; insignificant to the nodal operating systems; and highly significant to the internodal communication subsystem. The significance of a given ratio may also differ for levels of abstraction within the computations at a particular system level—e.g., within some system level, there may be: object method invocations, to which the ratio is relatively insignificant; built on layered remote procedure calls, to which the ratio is rather significant; which in turn are built on (uniform, location-transparent) message passing, to which the ratio is quite insignificant. The significance of the nodal/internodal performance ratio to a computation—i.e., its degree of physical distribution—depends on intrinsic characteristics of the computation, essentially related to how autonomous the per-node components are, and on the programming model used to express the computation.

The application pull for physically distributed computing in real-time contexts is both involuntary and voluntary.

The most common involuntary motivation is that application assets (e.g., the telecommunications switching offices, the different processing stages of a manufacturing plant, the ships and aircraft of a battle group) are inherently spacially dispersed, and a real-time performance requirement does not permit the latency of the requisite number of communications which would be needed between those assets and a centralized computing facility.

A prominent voluntary reason for physical dispersal is survivability, in the sense of graceful degradation for continued availability of situation-specific functionality. For example, it may be more cost-effective to distribute—i.e., replicate and partition—a telecommunications operation system, or an air/space defense command system, than it is to attempt to implement a physically centralized one which is infallible or indestructible.

There is also a powerful contemporary technology push for physically distributed computing due to the rapid increases in microprocessor performance and decreases in cost. This too is both voluntary and involuntary—the latter is due to current primary memory subsystems being disproportionately slower than processors, making clustered multicomputers attractive; regular topology ar-

ray style multicomputers also exhibit physically distributed computing properties as well.

Because of their physical dispersal, most distributed real-time computing systems are "loosely coupled" via i/o communication (employing links, buses, rings, switching networks, etc.), without directly shared primary memory. This generally results in variable communication latencies (regardless of how high the bandwidth) which are long with respect to local primary memory access times. The nature, locations, and availability of the applications' physical assets often limit the system's viability if it becomes partitioned (unlike, for example, a network of workstations), so these internode communication paths are frequently redundant and physically separated to reduce the probability of that happening.

A typical non-real-time distributed computing system—fitting the workstation model, such as [1])—is a network of nodes, each having an autonomous user executing unrelated local applications with statically hierarchical two-party (e.g., client/server) inter-relationships, supported by user-explicit resource management which is primarily centralized per node. In contrast, a real-time distributed computer system is mission-oriented—i.e., the entire system is dedicated to accomplishing a specific purpose through the cooperative execution of one or more applications distributed across its nodes. Thus, there is more incentive and likelihood for the nodes to have dynamically peer-structured multiparty inter-relationships at the application and OS layers.[1]

Real-time distributed computing applications are usually at a supervisory level, which means that their two primary functions are generally distributed system-wide resource management, and mission management. The former function is the application-specific portion (at least) of the distributed real-time execution environment, which augments the real-time (centralized or distributed) OS to compose the constituent subsystems into a coherent whole that is cost-effective to program and deploy for the intended mission(s). The latter function then utilizes this virtualized computing system to conduct some particular mission. It is far more probable than in a non-real-time dedicated-function system (e.g., for accounting) that the mission's approaches and even objectives are highly dependent on the current external (application environment) and internal (system resource) situation. Many real-time distributed computing applications are subject to great uncertainties at both the mission and system levels ("the fog of war" [3] is the extreme but most obvious example.

These application characteristics, combined with the laws of physics involved in distribution, results in the predominant portion of the supervisory level computing system's run-time behavior being unavoidably asynchronous, dynamic, and non-deterministic.[2] Therefore, even though most of the application results have (hard and soft) real-time constraints, it is not always possible for all of them to be optimally satisfied, nor to exactly know in advance which ones will be [4].

Nonetheless, real-time distributed computing applications and systems are usually *mission-critical*, meaning that the degree of mission success is strongly correlated with the extent to which the overall system can achieve the maximal dependability—regarding real-time effectiveness, survivability, and safety—possible given the resources that are available (in the general sense—e.g., operational, suitable, uncommitted, or affordable). The dependability of lower-level subsystems may be either necessary for mission-critical functions (e.g., digital avionics flight control keeping the aircraft aloft), or part of the uncertainty to be tolerated at the system and mission levels (e.g., communications, weapons); but it is not sufficient (e.g., a flying aircraft which cannot perform its mission is wasting resources and creating risks).

1. *Logically* distributed computing can be defined in terms of the kinds and degrees of multilateral resource management [2].

2. Non-determinism includes stochastic activity as a subset (e.g., Petri nets are the former but not the latter); some non-determinism in distributed real-time computing systems may have, or be usefully considered to have, a probability measure—this can permit more tractable resource management algorithms.

This implies the need for *best-effort* real-time resource management—accommodating dynamic and non-deterministic resource dependencies, concurrency, overloads, and complex (e.g., partial, bursty) faults/errors/failures, in a robust, adaptable way so as to undertake that as many as possible of the most important results are as correct in both the time and value domains as possible under the current mission and resource conditions [5][6][7]. It also entails offering the application user opportunities to at least participate in, if not control, the requisite resource management negotiations and compromises by adjusting his mission objectives and expectations to fit the circumstances, or changing the circumstances (constraints, resources), if either alternative is possible.

The option of best-effort resource management makes possible a choice between very firm á priori assurances of exact behavior in a limited number of highly specific resource and mission situations (as offered by static, highly predictable real-time technology), versus weaker assurances of probable behavior over a much wider range of circumstances. Examples of applications which seem to call naturally for either highly predictable or best-effort resource management come immediately to mind, as do others where the decision is more obviously a value judgement regarding risk and cost management under the exigencies of the situation.[3].

Virtually all such real-time reconciliation of uncertainty and dependability at the system and mission levels has historically depended solely on the talent and expertise of the system's human operators—e.g., in the control rooms of factories and plants, in aircraft cockpits. Increasingly, the complexity and pace of the systems' missions, and the number, complexity, and distribution of their resources, cause cognitive overload which requires that these operators receive more support in this respect from the computing system itself. Application software cannot solely bear this responsibility because the effectiveness of any resource management policy—especially real-time ones—depends on how consistently it is applied to all resources down to the lowest layer hardware and software. Moreover, best-effort policies place special demands on almost all the OS facilities.

The role of traditional real-time computers and OS's has been limited to being automatons in low-level sampled-data subsystems, where this contention between accommodating uncertainty and ensuring dependability does not arise. There, the premise is that the application and system's behavior is (or can be made) highly predictable, allowing extensive á priori knowledge about load and communication timing, exceptions, dependencies, and conflicts. Standard real-time theory and practice is to attempt to exploit such information with static techniques which aspire to provide guarantees about application and system behavior (not just the ends to be achieved, but even the exact means by which they are achieved)—but only under a small number of rigidly constrained, and often unrealistic, mission and resource conditions which are anticipated and accommodated in advance [9]. The classic real-time static, deterministic mindset and methodology constitute a simple special case, usually adequate for its intended domain, which does not scale up to distributed real-time systems.[4]

Therefore, one essential aspect of the research underlying the Alpha kernel was an improved understanding of "real-time" resource management.

3. It is instructive and enlightening to consider this issue in light of the many conclusive demonstrations by cognitive scientists of the ubiquitous human trait to miscalculate risks: for example, because we tend to be probability-blind near the extremes, we judge the annihilation of a risk very differently from the "mere" reduction of that risk, even when the latter diminishes the risk by a far greater degree [8].

4. There are analogous paradigm shifts in nature for larger, more complex systems—e.g.: in physics, where Newton's view of gravity as a force was generalized by Einstein as space/time curvature [10]; and in biology, where higher animals are more complex because they are larger, rather than conversely [11].

3. Real-Time in Alpha

The classical "hard/soft real-time" dichotomy has proven to be unnecessarily confusing and limiting, even for the centralized context in which it arose. We created the Benefit Accrual Model [12][13] to overcome the limitations of the classical one, and especially to facilitate the expansion of real-time computing into distributed systems.This model generalizes Jensen's notion of time-value function resource scheduling [14] (high performance architectural support for them was also initially explored [15]). Time-value function and best-effort scheduling has long been a research topic of the Archons project [16][17][18], and subsequently is being studied by others (e.g., [19][20]); its first OS implementation was in Alpha [17], and it has also appeared elsewhere, such as in Mach [21].

We regard a computation to be a *real-time* one if and only if it has a prescribed completion time constraint representing its urgency—i.e., time criticality—which is one of its acceptability criteria. Therefore, an OS is real-time to the degree that it explicitly (whether statically or dynamically) manages resources with the objective of application (and consequently its own) computations meeting their time constraint acceptability criteria. Thus, according to our definition of a real-time system, physical time, whether absolute or relative, is part of the system's logic—analogous to faults being states in a fault-tolerant system.

A computing system may meet its time constraint criteria without explicitly managing its resources to do so—by being endowed with excess resources (e.g., MS-DOS on a Cray Y-MP is "real fast" rather than real-time), or by good fortune—in which cases the system may fairly be considered to *operate in real-time* (and is not of interest to us).

In the classical "soft" real-time perspective, computation completion time constraints are usually not explicitly employed for scheduling; and in the corresponding "hard" real-time view, activity completion time constraints are defined as deadlines. In our Benefit Accrual Model, time constraints are both explicit and richer to delineate and encompass the continuum from "soft" to deadlines. They are represented with two primary components: the expression of the *benefit* to the system that the results yield, individually and collectively, as a function of their completion times; and application-specific predicates for acceptability optimization criteria based on *accruing* benefit from the results—see Figure 1.

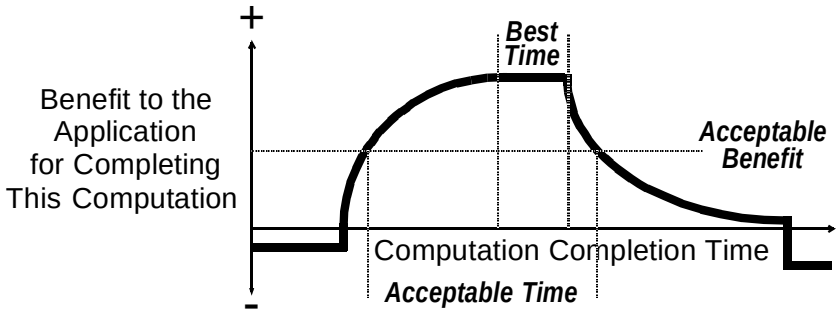

Figure 1: Benefit Function

Alpha's principal real-time strategy is to schedule all resources—both physical (such as processor cycles, secondary storage, communication paths) and logical (such as virtual memory, local synchronizers, and transactions)—according to real-time constraints using the Benefit Accrual Model described in the preceeding subsection.

A uniform approach to resource scheduling allows each α-thread itself to control all the resources it utilizes anywhere in the system—e.g., across nodes, and from user through to devices (such as

performing disk, network, and sensor/actuator accesses). The resulting continuity of the α-thread's time and importance (among other) attributes, together with appropriate scheduling algorithms, ensures coherent maintenance of real-time behavior.

Alpha separates resource scheduling into application-specific policies—e.g., defining optimality criteria and overload management—and general mechanisms for carrying out these policies. The mechanisms, together with a policy module interface, are part of the kernel. There are no restrictions on the kind, or number of, scheduling policies; obviously the parameters, such as time constraints and importance, must be interpreted consistently over a domain of execution, but there may be multiple such domains. The policies may use time directly, as with deadline algorithms, or indirectly, as with periodic-based fixed priority algorithms (e.g., rate-monotonic), or not at all, as with round robin.

For Alpha's context (notably characterized by aperiodicity and overloads in a distributed system), we conceived a new class of *best-effort* real-time scheduling policies [4]. Such policies are non-stochastically non-deterministic according to our taxonomy of scheduling [22].

Best-effort scheduling policies utilize more application-supplied information than is usual, and place specific requirements on the kind of scheduling mechanisms that must be provided. Obviously, resource scheduling which employs more application-supplied information, such as benefit functions, exacts a higher price than when little such information (e.g., static priority) or no information (e.g., round robin) is used. That price must be affordable with respect to the correctness and performance gained in comparison with simpler, less expensive, scheduling techniques.

The effectiveness and cost of a representative best-effort benefit accrual algorithm have been studied by simulation [6][7] and measurement [23][24]. The results demonstrate that this kind of scheduling is capable of successfully accruing greater value than the widely used algorithms—e.g., round robin and shortest-processing-time-first (both non-real-time algorithms), static priority (the most common real-time algorithm), and closest-deadline-first—for loads characteristic of Alpha's intended environment (at least). The scheduling cost per thread and per scheduling decision depend on the specific algorithm and on the implementation of time-value function representation and evaluation. The conclusion of these initial studies was that it is feasible to design and implement best-effort benefit accrual algorithms which provide a greater return to the application for resource investment than if some of those resources were available to the application itself because of lower-cost scheduling. Further experiments, together with research on analytical characterization of the performance of best-effort and benefit function scheduling, are taking place.

If desired, a large part of the price can be paid with the cheap currency of hardware: in multiprocessor nodes, a processor can be statically or dynamically assigned to evaluating the time-value functions (as is done in Alpha Release 1 [25] and Release 2 [26], respectively); or a special-purpose hardware accelerator, analogous to a floating point co-processor, could be employed.

4. Distribution in Alpha

Alpha is a distributed kernel which provides for coherent distributed programming of not only application software but also of the OS itself. It exports a new programming model which appears to be well suited for writing real-time distributed programs. Consequently, it also provides mechanisms having the objective of supporting a full range of client layer trans-node resource management policies; these policies are clients of the kernel and so are not discussed here.

Alpha provides a new kernel programming model because extant ones (cf. [27]) were deemed inappropriate for Alpha's objectives in various ways. For example, message passing and (direct read/write) distributed shared (virtual) memory (e.g., [28]) were rejected as being too low level (i.e., unstructured) for cost-effectively writing real-time distributed programs; distributed shared memo-

ry also suffers from implementation difficulties (e.g., the cost of synchronizing dirty pages). The conventional layered remote procedure call model is client/server oriented and thus imposes disadvantageous server-centric concurrency control; contemporary implementations also have insufficient transparency of physical distribution. Distributed data structures, such as Linda's Tuple Space [29], suffer from access sequencing design decisions—such as non-determinism and fairness—which render them unsuitable for time-constraint ordered tasks. Language-dependent models, like Argus [30], potentially offer significant advantages but are not acceptable in Alpha's overall user community. Virtually all of the few real-time distributed OS programming models are intended only for strictly deterministic systems (e.g., [31][32]).

Alpha's native programming model is provided at the kernel interface so the OS itself, as well as the applications, can employ real-time distributed programming. The OS layer can augment the kernel-supplied programming model in application-specific ways, or substitute an alternative one (e.g., POSIX, although full UNIX compatibility has never been an Alpha goal and therefore may be inefficient).

Alpha's kernel presents its clients with a coherent computer system which is composed in a reliable, network-transparent fashion of an indeterminate number of physical nodes. Its principle abstractions are objects, operation invocations, and distributed threads; these are augmented by others, particularly for exceptions and concurrency control [33][34].

4.1 Objects

In Alpha, objects are passive abstract data types (code plus data) in which there may be any number of concurrently executing activities (Alpha's *distributed threads*); semaphore and lock primitives are provided for the construction of whatever local synchronization is desired. Each instance of an Alpha client level object has a private address space to enforce encapsulation; the resulting safety improvement is judged to generally be worth the higher operation invocation cost in Alpha's application environments (but if performance dictates, objects may be placed in the kernel, as discussed in the Invocation subsection below).

An instance of an object exists entirely on a single node. Alpha's kernel supports dynamic object migration among nodes. Kernel mechanisms allow objects to be transparently replicated on different nodes, and accessed and updated according to application-specific policies.

Alpha objects are intended to normally be of moderate number and size—e.g., 100 to 10,000 lines of code—as dictated by the implemented cost of object operation invocation. The kernel defines a suite of standard operations that are inherited by all objects, and these can be overloaded.

Objects and their operations are identified by system-protected capabilities which provide a network location-independent space of unique names. Capabilities can be passed as invocation parameters.

An object may be declared permanent, which causes a non-volatile representation of its state to be placed in a local crash-resistent secondary storage subsystem, the mechanisms of which are normally (but not necessarily) resident in the kernel. These mechanisms also support application-specific atomic transaction-controlled updates to an object's permanent representation, which are performed in real-time—i.e., scheduled according to the real-time constraints of the corresponding distributed threads. This necessitates that Alpha take an integrated approach to managing resources in accordance with both the time-related, and the particular logical dependency, constraints which define execution correctness and data consistency; most other OS's (whether or not they are real-time and whether or not they are distributed) deal with these two kinds of constraints separately, if at all. Permanent objects obviate the need for a traditional file system in many applications, but any desired file system organization and semantics can readily be provided by client (system or application) layer policies.

In the interests of generality, Alpha's kernel views the universe of objects to be flat; any structure is added by higher software layers.

4.2 Operation Invocations

The invocation of an operation (method) on an object is the vehicle for all interactions in the system, including OS calls. Distributed threads (see the next subsection) are end-to-end computations (not processes or threads confined to an address space) which extend from object to object via invocations. Thus, operation invocation has synchronous request/reply semantics (similar to RPC); operations are block structured.

It is straight-forward to augment (or subvert the intent of) Alpha's synchronous programming model by constructing alternative asynchronous computational semantics on the native mechanisms—e.g., message sends which don't wait, and calls which spawn a concurrent activity that might not return. Asynchronous IPC (like assembly language programming) has a long history of use and staunch supporters in real-time computing. But the same effects can be accomplished in a better behaved manner by proper use of Alpha's model. Its kernel-level invocations are deliberately synchronous (and its threads distributed) because employing asynchrony is generally not straightforward, particularly for handling all the kinds of concurrency and exception cases which happen in a distributed real-time system. Even non-real-time, centralized multiprocessing OS's whose native IPC mechanisms are asynchronous often seek to improve the intellectual manageability of client programming by also providing layered synchronous facilities. Mach provides examples of this: MIG [35] is used not just for RPC but also sometimes for local IPC, and is the only IPC facility provided in a fault tolerant system built on Mach [36]; an approach to transparent recovery which does use Mach's asynchronous messages is significantly complicated by them [37]. Similarly, the asynchronous message passing communication hardware of large multicomputers is often abstracted into a more productive synchronous programming model with software development tools [38]. Asynchronous RPC was removed from Amoeba 2.0 as having been "a truly dreadful decision" and "impossible to program correctly" [39]. Asynchronous IPC is also highly problematic for attaining the TCSEC B3 level of assurance for multilevel security in OS's (e.g., in Trusted Mach [40]).

Invocation parameters are passed into the invoked object's domain on invocation, and when the invocation is complete, return parameters are passed back to the invoking object's domain. All invocation (request and reply) parameters, except capabilities, are passed by value on the current frame and stack for this invocation by the distributed thread; each distributed thread has its own stack and cannot access the stack of another. Handling bulk data (e.g., [41]) does not seem to be a typical requirement in system integration and mission management applications (a programmer-transparent implementation enhancement facilitates movement by value of large parameters within a node); however, asynchronous bulk data movement can be performed as a kernel client layer service without changing the programming model. We consider procedural parameters contrary to the spirit of object oriented systems. Alpha does not presently deal with the topic of parameter representation conversion which arises among heterogeneous nodes; that problem receives wide attention elsewhere (unlike most of those we are currently focusing on), and since Alpha does not require an especially unique solution, we will adapt one when necessary.

Invocation, not simply message passing, is a fundamental kernel facility of Alpha. Consequently, objects may be placed within the kernel address space for performance improvements. Of course, if they seek further speedup by directly accessing kernel data structures, that forecloses the (sometimes desirable) option of moving them back out of the kernel into client space.

We think of each inter-node invocation as creating a *segment* of that distributed thread. Invocation masks the effects of thread segmentation with unusually strong semantics for independence and transparency of physical distribution [42].

Alpha's kernel performs implicit binding at the time of each invocation (utilizing a protocol, rather than a centralized name server such as the NCS Location Broker [43], for locating the target object). The kernel also includes provisions to optionally perform explicit binding; this is for optimizing performance in relatively static cases (e.g., due to specially located resources), and for other purposes such as testing and failure recovery.

Communication errors are handled by message protocols which may be realized as kernel or client objects. The various motivations for reliable messages being entirely client level functionality (e.g., the microkernel arguments and the end-to-end argument [44]) must be balanced against the acceptability constraints of the particular real-time system. Alpha's communication subsystem incorporates an approach to a general framework and mechanisms for implementing communication protocols, due to the requirement for problem-specificity imposed by real-time requirements [45]; future versions may substitute corresponding concepts and techniques from the *x*-Kernel [46].

Alpha provides orphan detection (presently under the usual assumption of fail-stop nodes) and elimination, at any time (even on a distributed thread which is already undergoing orphan elimination), and in a decentralized manner [45]. The technique employed requires active tracking of the progress of each distributed thread by the Alpha instance on the node where that thread is rooted. However, any orphaned activity can be successfully detected and eliminated, without requiring significantly more complex mechanisms such as transactions or distributed clocks [47]**.** This technique also allows dynamic trade-offs of communication bandwidth and processing against orphan detection latency. The standard Alpha configuration is for orphan detection and elimination to be kernel functionality, although it can alternatively be implemented in client space if desired·

Invocations may fail for various reasons, such as protection violation, bad parameters, node failure, machine exception, time constraint expiration, or transaction abort. The failure semantics of invocation instances in a real-time distributed system must be application-specific, so Alpha's kernel includes additional mechanisms for defining them; at-most-once is the default. See the subsection below on exceptions.

4.3 Distributed Threads

An Alpha *distributed thread* (α-*thread*) [17] is the locus of control point movement among objects via operation invocation, as shown in Figure 2. It is a distributed computation which transparently and reliably spans physical nodes, contrary to how conventional threads (conceived as lightweight processes) are confined to a single address space in most other recent OS's such as Mach [48] and Chorus [49]; however, Clouds [50] employs a thread model similar to Alpha's.

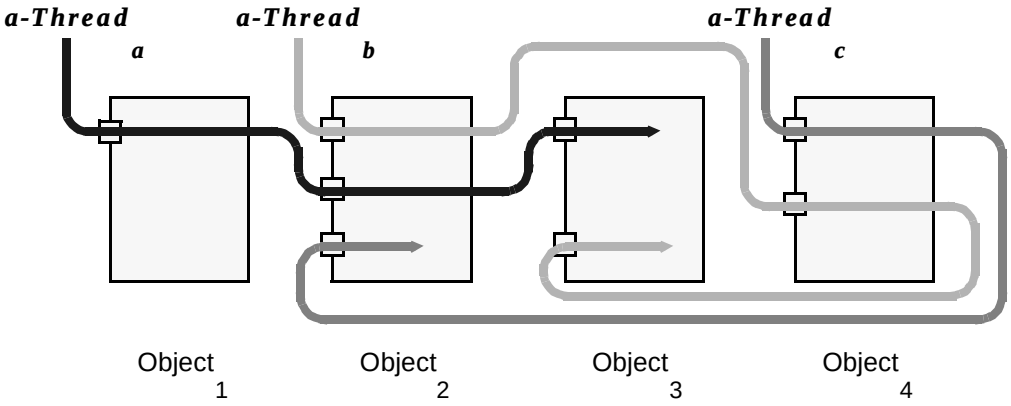

Figure 2: Alpha's *Distributed Threads* (α-*Threads*)

An α-thread carries parameters and other attributes related to the nature, state, and service re-

quirements of the computation it represents. An α-thread's attributes may be modified and accumulated in a nested fashion as it executes operations within objects (illustrated in Figure 3). Unlike how

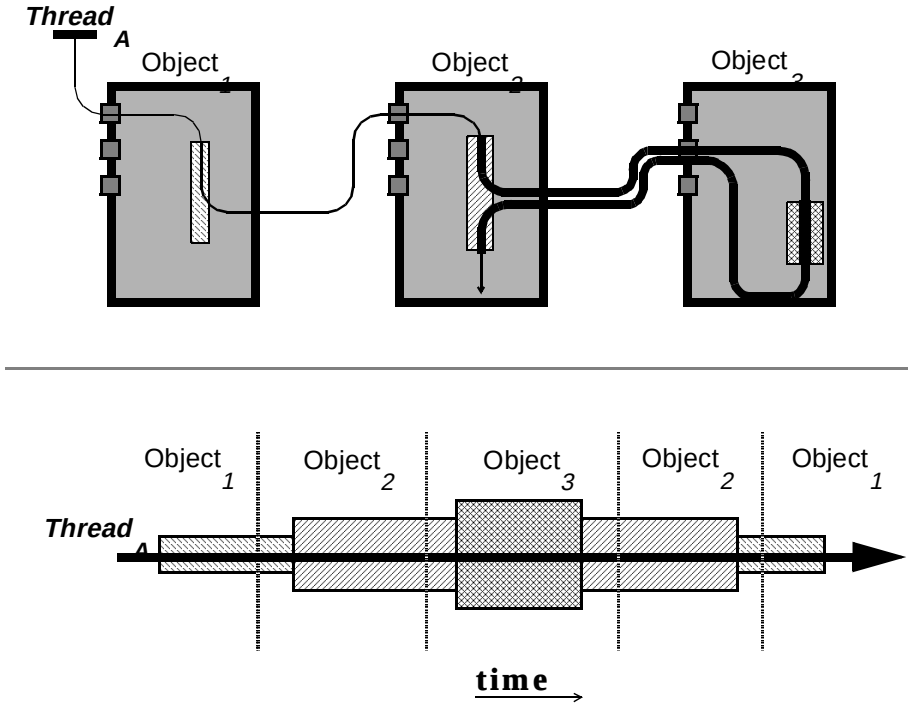

Figure 3: α-thread Attribute Accumulation

RPC or message passing are employed in other OS's, these attributes are utilized by Alpha's kernel and its clients as a basis for performing real-time, system-wide, decentralized resource management.

α-threads are the unit of schedulability, and are fully pre-emptable, even those executing within the kernel. Thus, when the scheduling subsystem detects that there is a ready α-thread whose execution is more likely to increase the accrued benefit than the one currently running, the executing α-thread can be pre-empted by the waiting one. The pre-emption costs and expected completion time of the lower benefit-accrual α-thread are taken into account when making this decision. In addition, Alpha offers scheduling algorithms which explicitly deal with the various kinds of resource dependencies and conflicts among α-threads, and if appropriate, they roll forward or roll back a lower benefit-accrual α-thread which is blocking a higher one [7]. The fully pre-emptable and multithreaded design of Alpha's kernel facilitates real-time behavior and allows symmetric multiprocessing within the kernel itself as well as within its clients.

4.4 Exceptions

Every α-thread is subject to exceptions—an event that interrupts the α-thread's normal execution flow. With respect to an α-thread's execution, an exception may be synchronous (e.g., a machine check) or asynchronous (e.g., a real-time constraint expiration, transaction abort, α-thread break). The kernel's exception handling mechanisms treat synchronous and asynchronous exceptions uniformly.

Alpha's kernel provides exception handling mechanisms defined in terms of kernel-provided abstractions; these language-independent mechanisms can be used by the OS and language run-time systems to construct appropriate exception handling policies, which clients may, in turn, use to es-

tablish application-specific exception handlers (which, for example, retry, perform compensatory actions, or utilize the results attained prior to occurrence of the exception). The mechanisms permit applications to define handlers for the core set of exception types defined by the kernel, and also to define their own exception types and handlers for them.

The mechanism for specifying exception handlers is the e*xception block*, a block-structured construct that complements the block-oriented nature of invocations. The `BEGIN` operation for exception blocks opens a scope of execution during which its parameters define the exception handlers to be used for the specified exception types while the α-thread is executing within that block. The `END` operation closes the inner-most open exception block. Like other α-thread attributes, exception blocks may be nested and exception block scoping is dynamic. The exception handling attributes are protected by the kernel, so that subsequent application errors cannot corrupt them.

When an exception of a particular type occurs, control of the α-thread is moved to the handler specified by the inner-most exception block that defines a handler for exceptions of that type. Because Alpha's kernel is fully pre-emptable, an exception may force an α-thread out of the kernel, at an arbitrary point (even if it is blocked), to perform exception handling. So, in addition to any user-defined exception blocks, the kernel treats each operation defined on an object as an implicit exception block. The kernel-defined handlers for these implicit blocks perform only the simple clean-up operations necessary to ensure that the kernel will retain a minimum degree of internal consistency (i.e., it will neither leak resources, nor fail due to inconsistent internal data structures). The existence of this implicit block also ensures that exception blocks opened in one object will not be closed in another (i.e., exception blocks must nest correctly within an object).

An α-thread always handles its own exceptions, preserving the correspondence between the α-thread and the computation it is performing. Following the occurrence of an exception, the kernel adjusts the attributes of the α-thread so that each exception handler is executed with attributes appropriate for the α-thread exception block at that point—among other things, this ensures that the proper scheduling parameters are associated with the exception handling.

The occurrence of a single exception may require multiple levels of exception handling to be performed. An example is a real-time constraint expiration exception, which is not discharged until the exception block level at which the real-time constraint was established is reached. Another example is the elimination of an orphan α-thread segment, where the exception is not discharged until the segment is eliminated. In such cases, exception handlers are executed in order from inner-most to outer-most until the exception is discharged.

If exception processing spans multiple invocations, all invocation frames of the α-thread except the head will be waiting for an invocation to complete. System-level interface libraries can take advantage of this fact to simplify application-level exception processing in these cases.

4.5 Transactions

Transactions are useful for a wide variety of integrity purposes, including the optional extension, when needed, of invocation semantics to zero/one (e.g., [51]). Of particular interest is that Alpha promotes (but is not limited to) a transactional approach to trans-node concurrency control[5] so that collectively α-threads behave "correctly," as defined by the application, and so that distributed (both replicated and partitioned) data remains mutually "consistent," as defined by the application [52]. The many advantages of this include permitting remote invocations to pass mutable parameters by value (which thus constitute shared state), while avoiding the limitations of conventional

5. In distributed systems, synchronization is generally achieved through maintaining an ordering of events, rather than through mutual exclusion as in centralized systems. We do not consider that sending messages to a centralized synchronization entity is consonant with the objectives of distribution.

server-centric concurrency control in network-style distributed systems. Transactions from the database context cannot be simply transplanted into an OS—this is particularly true for real-time systems because they are integrated into the physical nature of the application.

One general problem is that traditional transactions bundle the properties of atomicity, permanence, and serializability together at one (high) performance price. Instead, Alpha's kernel provides transaction mechanisms for atomicity, permanence, and application-specific concurrency control individually; these can be selected and combined at higher (OS and application) layers according to a wide range of different transaction policies whose cost is proportional to their power.

In real-time systems, permanence is not universally desirable: some transactions update data that is relevant only to the local node for this incarnation; and in many cases, the physical world maintains the true state (as related to the system by the sensor subsystems) that is only cached or approximated by the data manipulated by transactions.

Conventional transactions define correctness as serializability, which limits concurrency and thus performance [53]. Alpha's mechanistic technique encourages the use of application-specific information in non-serializable transactions. This allows optimized correctness through customized commit and abort handling: transactions can commit and allow other transactions to observe their results with no ill effect for an arbitrary period of time; their abort processing can also be deferred for an arbitrary period of time (unless there are other mitigating circumstances). Traditional recovery techniques such as rollback and redo, and those requiring the client applications to be deterministic or idempotent (e.g., stateless) [54][55], are not always germane in real-time contexts. Furthermore, performance can be improved through cooperation among non-serializable transactions [56].

The second major limitation of conventional transactions is that they do not have and use information about application result real-time constraints. They are scheduled according to different criteria (e.g., serializability) than are the tasks (α-threads in Alpha's case); they employ locking mechanisms (e.g., time stamps) unrelated to task (α-thread) scheduling; and the time required to acquire and release resources, as well as the time required to commit and abort transactions, is potentially unbounded. To overcome these limitations, Alpha's transactions are real-time, most importantly meaning that they are scheduled according to same application real-time constraints and policies as are all other resources.

4.6 Alpha System Architectures

A distributed OS could impose or accommodate a variety of possible OS configurations and thus system architectures [4]; Alpha is primarily intended for three of these.

The purest form of a distributed OS is for it to be the only OS in the system—native on all nodes—in which case, it must provide local OS functionality as well, and be cost-effective for both local (centralized) and distributed applications. The nodes, and their interconnection, must have sufficient resources to support both local and distributed computations. It is difficult for such an OS to accommodate backward compatibility with extant local OS or application software, but it is the cleanest and most coherent approach when there is the freedom to create an entire system from scratch (as is often the case in real-time applications, especially distributed ones).

The second system architecture of interest to us is for the distributed OS to be native on its own interconnected hardware nodes, forming a *global OS* (GOS) subsystem This necessitates that it provide full local OS functionality as well, although not necessarily in a manner which is most cost-effective for low-level, sampled-data real-time applications. The GOS subsystem nodes physically interface with the local nodes and OS's—which constitute the low-level real-time subsystems—via the GOS subsystem interconnect and/or a system level interconnection. This case offers: superior performance due to local/global hardware (node and interconnect) concurrency; compatibility with heterogeneous and pre-existing local subsystems (OS's and applications); major logistical benefits

from the relative independence of local and global OS and application changes.

The third alternative system architecture for Alpha is for there to be distributed and local OS's which are separate but co-resident on the local nodal hardware. On uniprocessor nodes, co-residency would require something like a virtual machine monitor to create an illusion of two or more processors, which entails overhead (that may be affordable because of the performance of contemporary microprocessors). On multiprocessor nodes, co-residency can be relatively easy and highly effective—Alpha can (and often does) co-exist and interoperate with UNIX on any or all of the system's multiprocessor nodes (as shown in Figure 4), thus making available to Alpha applications all the

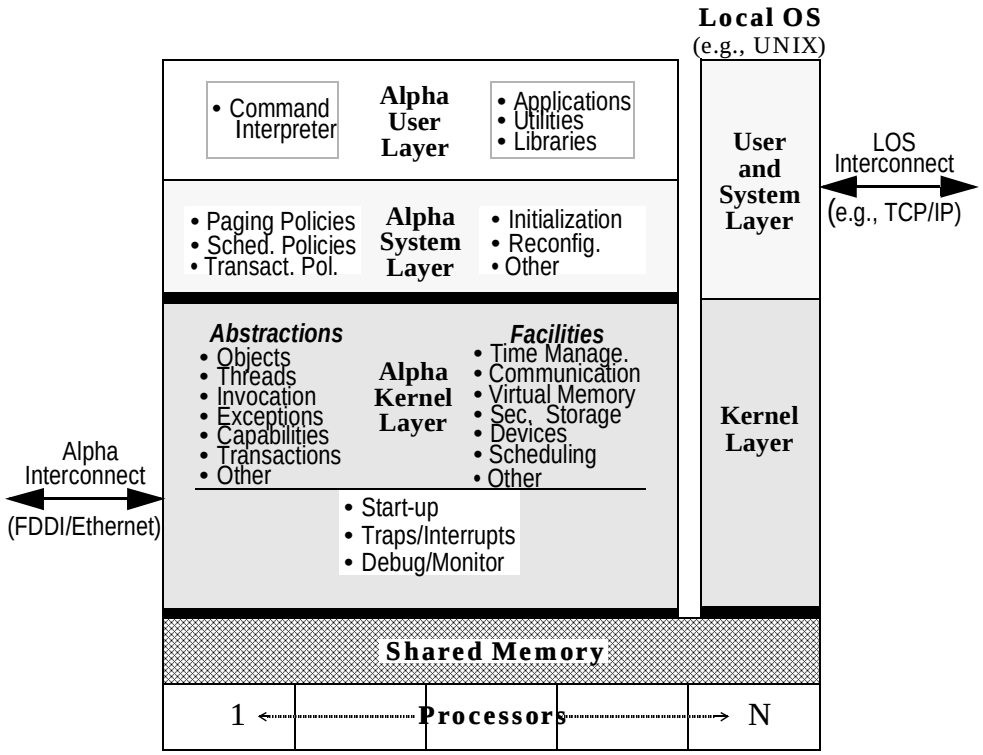

Figure 4: Alpha Co-Resident With UNIX On Multiprocessor Nodes

non- (or less) real-time functionality of UNIX (such as GUI's, ISO protocols, and software development facilities) [57]. With either configuration, the internode connection must be able to support both distributed and local OS's, or there must be separate interconnection structures for each type.

5. Multilevel Security in Alpha

The construction of multilevel secure (B3 and higher) [58], distributed, real-time systems is of great interest to a large part of Alpha's prospective user community, and thus is an area of active research within the Alpha project [59]. There are many inherent conflicts between the requirements of real-time and B3 security, including (but not limited to): covert timing channels arising from the real-time scheduling algorithms; covert storage channels due to resource sharing and contention; the potential for malicious denial of service by untrusted applications improperly asserting great urgency and importance; and predictability of security enforcement behavior. These conflicts are likely to require appropriately authorized, situation-specific, dynamic trade-offs between various security and real-time characteristics.

6. Architectural Lessons Learned From Alpha

The following synopsizes some of the important lessons we believe that we have learned from our experiences with Alpha's architecture in comparison with that of others, such as Mach 3.0.

6.1 Kernel Support for Distributed Threads

It is common to find RPC services implemented as a layer on top of asynchronous message passing facilities. This layering usually involves multiple scheduling events, complex RPC stubs for argument marshalling, multiple context changes, and the consequent loss of the client's identity, time-constraint, and other attributes. Mach, as well as some other OS's, attempts to overcome some of these deficiencies by providing subsets of the message passing service that are optimized for RPC [60]. In the case of Mach, only simple messages are optimized, but messages with capabilities (port rights) or out-of-line data do not benefit from the optimizations. These optimizations reduce the number of scheduling interactions and system calls necessary to implement RPC, but identity, scheduling information (e.g., priority) and other attributes are not propagated. In contrast, when an Alpha distributed thread moves from object to object, its time constraints and other properties remain in effect. Alpha's kernel is fully pre-emptable, and every effort is made to run the most important ready thread whether it is executing in client space or kernel space.

Timely service interruption processing is essential to Alpha's strategy for scheduling overload situations. Alpha's exception model explicitly takes into account orphans and distributed exceptions. The need for this functionality is not unique to real-time systems; UNIX and POSIX compatible systems also must support interruptible system calls. Orphan detection and elimination is typically not provided by layered RPC facilities.

These limitations of layered RPC facilities make building a distributed real-time RPC facility problematic and inefficient. Recently published work suggests that high performance RPC is best obtained with RPC-specific kernel assistance [60][61][62][63].

Multi-server operating systems have many of the characteristics of distributed applications even if all the servers reside on a single node. The client process communicates with the OS server(s) via IPC. In a standard implementation of UNIX, when an application invokes a system service the client thread of control moves from the user application context into the operating system. When the request for service is completed the thread returns to user space. A variety of attributes follow the thread from user space to the kernel including identity, working directory, quotas, etc. The kernel uses these attributes to track and manage resource consumption, provide interruptible system calls and insure security. The interaction of user applications with standard operating systems is very reminiscent of distributed threads. Mimicking the semantics of "legacy" operating systems, UNIX in particular, with a collection of servers is complicated by microkernels that do not provide sufficient support for distributed programming.

Auditing and authentication forwarding are significant problems for secure distributed systems. Changing identity or subjects during a request for service makes it difficult to associate the server actions or resources with the client responsible for the request for service. This association is important for both authentication and auditing. Distributed threads facilitate this aspect of security by preserving the identity of the distributed computation.

6.2 Dynamic, Adaptive Thread Management

Most kernels, such as Mach and Chorus, provide a threads abstraction that associates each thread with a single task. By default, Mach thread management is static. If servers are over-subscribed, then requests block, regardless of whether there are computation resources available. If the server

is under-subscribed, then kernel resources such as process control blocks and other kernel data structures are reserved but under-utilized.Dynamic or adaptive thread management is the responsibility the application designer. Experience has shown that application level solutions to thread management for distributed systems tend to be complex, inefficient and prone to error.

Thread management based on global (inter-task) resource usage and requirements is difficult and not possible without compromising security. Alpha threads do not prevent applications from controlling the number of extant threads, however the default behavior is to create threads dynamically, on an as-needed basis. The kernel, with its knowledge of available resources, has the primary responsibility for balancing the computation demands against the available resources.

It has been argued that thread management and RPC layers can be constructed in user space that provide most of the advantages of distributed threads; this may be true, but the examples that the authors are familiar with are not convincing.

6.3 Protected Capabilities

Alpha capabilities are kernel protected and have context sensitive names, similar to Mach port rights and port names. Other systems, such as Chorus and Amoeba, provide unprotected capabilities. While the cost of invoking either type of capability seems roughly equal, our experience confirms the assertion that unprotected capabilities are somewhat less expensive to pass as parameters. However, unprotected capabilities are insufficient to build high trust systems [64][65]. Though early versions of Alpha associated capabilities with objects (similar to the way Mach associates ports and port rights with tasks), we found this awkward and inconvenient. Subsequent versions of Alpha permit capabilities to be associated with threads. Thread local capabilities simplify capability management. They enable the construction of secure subsystems and can be leveraged in other ways when building secure real-time systems [59].

6.4 Object Invocation Via Broadcast

Alpha uses broadcast protocols aggressively to locate and invoke remote objects. Each node maintains a list of objects that are local to the node—the object dictionary. When an object invocation is broadcast, each node receives the message and examines its dictionary to determine if the object being invoked resides locally. This results in simple object location protocols with a relatively constant time (a useful property for real-time applications). While the time required to locate the object is small, the broadcast processing overhead imposed on nodes can be significant. If the dictionary of objects is too large to fit in memory then it must be paged. Paging would add significantly to the total overhead and the variance of broadcasting object invocations.

Other OS's, and Mach in particular, have demonstrated that if the kernel "owns" capabilities, it is possible to track and cache the current location of any capability or object. Though Alpha currently broadcasts each remote object invocation, we have done a preliminary investigation into caching remote object location information as one means to reduce the number of object invocation generated broadcasts.

6.5 Separation of Policies from Mechanisms

The separation of policies from mechanisms is more than code words for "layered design." The design of mechanisms and policies is an approach to encapsulation and layering that results in relatively simple mechanisms suitable for the implementation of a variety of policies.

Our experience suggests that the kernel interface is not the only interesting mechanism/policy boundary. We have found that the creation of policy modules within the kernel—such as for scheduling, secondary storage, and communications—was of great value. The scheduling subsystem is not simply layered; a common set of mechanisms has been used to implement a number of signifi-

cantly different real-time and non-real-time scheduling policies. The encapsulation of these policies facilitated not only their development and maintenance but also their wholesale replacement.

7. Project History and Status

Alpha arose as the first systems effort of Jensen's Archons Project on new paradigms for real-time decentralized computer systems, which began in 1979 at Carnegie-Mellon University's Computer Science Department. Design of the Alpha OS itself was started in 1985 and the initial prototype ("Release 1") was operational at CMU in the Fall of 1987.

Alpha is a native kernel (i.e., on the bare hardware), which necessitates a great deal of effort on low level resource management. But much of that work involves fundamental issues in Alpha's design and implementation—the usual research approach of emulating an OS at the application level (typically on UNIX) would have introduced excessive real-time distortions due to the vast disparity between the programming model and structure of Alpha's kernel and those of UNIX. Alpha's first hardware base was multiprocessor nodes built from modified Sun Microsystems and other Multibus boards; the nodes were interconnected with Ethernet.

The principle goal of the Archons Project in general and its Alpha component in particular was both to create new concepts and techniques for real-time distributed OS's, and to validate those results through industrial as well as academic experimentation. The first step in that validation process was to augment our own personal experience with industrial real-time distributed computing systems by involving a user corporation early in the development of the initial Alpha prototype. Because Archons and Alpha were sponsored primarily by the DOD, and because the leading edge real-time computing problems and solutions always arise first in defense applications, we looked to the DOD contractor community for our initial industrial user partnership. We selected General Dynamics' Ft. Worth Division, exchanging application and OS technology in the highest bandwidth way—by exchanging people. Their C^3I group successfully wrote and demonstrated a real-time distributed air defense application on Alpha in 1987 [23], and their avionics organization intended to base the mission management OS of a planned new aircraft on Alpha's technology.

Once the proof of concept prototype was operational, we sought to begin transition of Alpha's technology into practice by establishing a relationship with a computer manufacturer. To further facilitate that transition, the leadership and then the staff of the Alpha project moved to industry in 1988. The intent is for Alpha to serve as a technology development vehicle—first for application-specific real-time distributed operating systems (e.g., for telecommunications, simulators and trainers, C^3I, combat systems) where extensive functionality (such as fault tolerance) and high real-time performance are of the utmost importance, no off-the-shelf products exist, and no standards are foreseeable for a number of years. Subsequently, the technology will be available for migration into other OS contexts. A second generation Alpha prototype design and implementation was delivered to several government and industry laboratories for experimental use; the first of these was installed in June 1990. The current version is initially available on MIPS R3000-based multiprocessor nodes interconnected by Ethernet; ports to other hardware are planned. Alpha is non-proprietary.

Alpha research is ongoing at CMU, and related research and technology development is also being conducted cooperatively with several other academic and industrial institutions. The Alpha project is also engaged in partnerships with a number of U.S. and European corporations and other organizations to develop experimental Alpha applications in the areas of telecommunications and defense systems.

8. References

[1] Anderson, D.P., D. Ferrari, P.V. Rangan, and S.-Y. Tzou, The DASH Project: Issues in the Design of Very Large Distributed Systems, report no. UCB/CSD 87/338, U. of CA/Berkeley EECS Department, January 1987.

[2] Jensen, E.D., *Decentralized Control*, Distributed Systems: An Advanced Course, Springer-Verlag, 1981.

[3] Clausewitz, C. von, On War, tr. by J.J. Graham, N. Trubner & Co. (London), 1873.

[4] Jensen, E.D., *Alpha: A Real-Time Decentralized Operating System for Mission-Oriented System Integration and Operatio*n, Proceedings of the Symposium on Computing Environments for Large, Complex Systems, University of Houston Research Institute for Computer and Information Sciences, November 1988.

[5] Jensen, E.D., C.D. Locke, and H. Tokuda, *A Time-Value Driven Scheduling Model for Real-Time Operating Systems*, Proceedings of the Symposium on Real-Time Systems, IEEE, November 1985.

[6] C.D. Locke, Best-Effort Decision Making for Real-Time Scheduling, Ph.D. Thesis, CMU-CS-86-134, Department of Computer Science, Carnegie Mellon University, 1986.

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

[8] Kahneman, D., P. Slovic, and A. Tversky (Ed.), Judgement Under Uncertainty: Heuristics and Biases, Cambridge University Press, 1982.

[9] Stankovic, J.A., Tutorial on Hard Real-Time Systems, IEEE, 1988.

[10] Einstein, A., Relativity: The Special and the General Theory, Crown, December 1916.

[11] Haldane, J.B.S., *On Being the Right Size*, Possible Worlds and Other Essays, Chatto and Windus, 1927.

[12] Jensen, E.D., *A Benefit Accrual Model of Real-Time*, Proceedings of the 10th IFAC Workshop on Distributed Computer Control Systems, September 1991.

[13] Jensen, E. D., *A Benefit Accrual Model of Real-Time*, submitted for publication, 1991.

[14] Stewart, B., Distributed Data Processing Technology, Interim Report, Honeywell Systems and Research Center, March 1977.

[15] Gouda, M.G., Y.H. Han, E.D. Jensen, W.D. Johnson, and R.Y. Kain, *Towards a Methodology for Distributed Computer System Design*, Proceedings of the Texas Conference on Computer Systems, IEEE, November 1977.

[16] Jensen, E.D., *The Archons Project: An Overview*, Proceedings of the International Symposium on Synchronization, Control, and Communication, Academic Press, 1983.

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

[18] Jensen, E.D., Alpha—A Non-Proprietary Operating System for Mission-Critical Real-Time Distributed Systems, Technical Report TR-89121, Concurrent Computer Corp., December 1989.

[19] Chen, K., *A Study on the Timeliness Property in Real-Time Systems*, Real-Time Systems, September 1991.

[20] Chen, K. and P. Muhlethaler, *Two Classes of Effective Heuristics for Time-Value Function Based Scheduling*, Proceedings of the 12th Real-Time System Symposium, IEEE, 1991.

[21] Tokuda, H, J.W. Wendorf, and H.Y. Wang, *Implementation of a Time-Driven Scheduler for*

Real-Time Operating Systems, Proceedings of the Real-Time Systems Symposium, IEEE, December 1987.

[22] Jensen, E. D., *A Taxonomy of Real-Time and Distributed Scheduling*, Course Notes, Distributed Real-Time Systems: Principles, Solutions, Standards, Paris, June 1991.

[23] Maynard, D.P., S.E. Shipman, R.K. Clark, J.D. Northcutt, R.B. Kegley, B.A. Zimmerman, and P.J. Keleher, An Example Real-Time Command, Control, and Battle Management Application for Alpha, Technical Report TR 88121, Archons Project, Computer Science Department, Carnegie-Mellon University, December 1988.

[24] Northcutt, J.D., R.K. Clark, D.P. Maynard, and J.E. Trull, Decentralized Real-Time Scheduling, Final Technical Report, Contract F33602-88-D-0027, School of Computer Science, Carnegie-Mellon University, February 1990.

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

[26] Reynolds, F.D., J.G. Hanko, and E.D. Jensen, Alpha Release 2 Preliminary System/Subsystem Description, Technical Report #88122, Concurrent Computer Corporation, December 1988.

[27] Tannenbaum, A.S. and R. van Renesse, *Distributed Operating Systems*, Computing Surveys, ACM, December 1985.

[28] Cheriton, D.R., H.A. Goosen, and P.D. Boyle, *Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture*, Computer, IEEE, February 1991.

[29] Kaashoek, M.F., H.E. Bal, and A.S. Tannenbaum, *Experience With The Distributed Data Structure Paradigm in Linda*, Proceedings of the Workshop on Distributed and Multiprocessor Systems, USENIX Association, October 1989.

[30] Liskov, B. and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust, Distributed Programs*, Transactions on Programming Languages and Systems, ACM, July 1983.

[31] Gudmundsson, O., D. Mosse, K-T. Ko, A.K. Agwrawala, and S.K. Tripathi, *MARUTI: A Platform for Hard Real-Time Applications*, Proceedings of the 1989 Workshop on Operating Systems for Mission-Critical Computing, IOS Press, 1992.

[32] Kopetz, H, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*, Micro, IEEE, February 1989.

[33] Northcutt, J.D. and R.K. Clark, The Alpha Operating System: Programming Model, Archons Project Technical Report TR-88021, Carnegie-Mellon University, February 1988.

[34] Northcutt, J.D., R.K. Clark, S.E. Shipman, D.P. Maynard, E. D. Jensen, F.D. Reynolds, and B. Dasarathy, *Threads: A Programming Construct for Reliable Real-Time Distributed Programming*, Proceedings of the International Conference on Parallel and Distributed Computing and Systems, International Society for Mini- and Micro-Computers, October 1990.

[35] Draves, R.P., M.B. Jones, and M.R. Thompson, MIG: The Mach Interface Generator, Internal Working Document, Computer Science Department, Carnegie-Mellon University, November 1989.

[36] Chen, R. and T.P. Ng, *Building a Fault-Tolerant System Based on Mach*, Proceedings of the USENIX Mach Workshop, October 1990.

[37] Goldberg, A., A. Gopal, K. Li, R. Strom, and D. Bacon, *Transparent Recovery of Mach Applications*, Proceedings of the USENIX Mach Workshop, October 1990.

[38] Wu, M.-Y. and D.D. Gajski, *Hypertool: A Programming Aid for Message-Passing Systems*,

Transactions on Parallel and Distributed Systems, IEEE, July 1990.

[39] Tannenbaum, A.S., R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Jansen, and Guido van Rossum, *Experiences With the Amoeba Distributed Operating System*, Communications of the ACM, December 1990.

[40] Branstad, M.A., H. Tajalli, F. Mayer, and D. Dalva, *Access Mediation in a Message-Passing Kernel*, Proceedings of the IEEE Symposium on Security and Privacy, May 1989.

[41] Gifford, D.K. and N. Glasser, *Remote Pipes and Procedures for Efficient Distributed Communication*, Transactions on Computer Systems, ACM, August 1988.

[42] Levy, E. and A. Silberschatz, *Distributed File Systems: Concepts and Examples*, Computing Surveys, ACM, December 1990.

[43] Apollo Computer Corp., Network Computing System Reference Manual, 1987.

[44] Saltzer, J.H., D.P. Reed, and D.D. Clark, *End-to-End Arguments in System Design*, Transactions on Computing Systems, ACM, November 1984.

[45] Northcutt, J.D., and R.K. Clark, The Alpha Operating System: Kernel Internals, Archons Project Technical Report #88051, Department of Computer Science, Carnegie Mellon University, May 1988.

[46] Hutchinson, N.C., and L.L. Peterson, *The x-Kernel: An Architecture for Implementing Network Protocols*, Transactions on Software Engineering, IEEE, January 1991.

[47] Herlihy, M.P., and M.S. McKendry, *Timestamp-Based Orphan Elimination*, Transactions on Software Engineering, IEEE, July 1989.

[48] Rashid, R., *Threads of a New System*, Unix Review, August 1986.

[49] Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herman, C. Kaiser, S. Langois, P. Leonard, and W. Neuhauser, CHORUS Distributed Operating Systems, CS/TR-88-7.8, Chorus Systemes, 1989.

[50] Dasgupta, P., R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. Leblanc, W.F. Appelbe, J.M. Berabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wilkenloh, *The Design and Implementation of the Clouds Distributed Operating System*, Computing Systems, USENIX, June 1988.

[51] Brown, M.R., K.M. Kolling, and E.A. Taft, *The Alpine File System*, Transactions on Computer Systems, ACM, November 1985.

[52] Jensen, E.D., *The Implications of Physical Dispersal on Operating Systems*, Proceedings of Informatica '82, Sarajevo, Yugoslavia, March 1982.

[53] Garcia-Molina, H., *Using Semantic Knowledge for Transaction Processing in a Distributed Database*, Transactions on Database Systems, ACM, June 1983.

[54] Borg, A. et al., *Fault Tolerance Under UNIX*, Transactions on Computer Systems, ACM, January 1989.

[55] Ravindran, K., and S.T. Canson, *Failure Transparency in Remote Procedure Calls*, Transactions on Computers, IEEE, August 1989.

[56] Sha, L., E.D. Jensen, R. Rashid, and J.D. Northcutt, *Distributed Co-Operating Processes and Transactions*, Proceedings of the ACM Symposium on Data Communication Protocols and Architectures, Mar. 1983, and Synchronization, Control, and Communication in Distributed Computing Systems, Academic Press, 1983.

[57] Vasilatos, N., *Partitioned Multiprocessors and the Existence of Heterogeneous Operating Systems*, Proceedings of the USENIX Winter 1991 Conference, January 1991.

[58] Gasser, M., Building A Secure Computer System, Van Nostrand Reinhold, 1988.

[59] Loepere, K.P., F.D. Reynolds, E.D. Jensen, and T.F. Lunt, *Security for Real-Time Systems*, Proceedings of the 13th National Computer Security Conference, October 1990.

[60] Draves, R., *A Revised IPC Interface*, Proceedings of the USENIX Mach Workshop, October 1990.

[61] Karger, P.A., Improving Security and Performance for Capability Systems, Technical Report No.149, Computer Laboratory, University of Cambridge, October 1988.

[62] Bershad, B.N., T. E. Anderson, E. D. Lazowska, H. M. Levy, *Lightweight Remote Procedure Call*, Transactions on Computer Systems, ACM, February 1990.

[63] Schroeder, M.D., and M. Burrows, *Performance of Firefly RPC*, Transactions on Computer Systems, ACM, February 1990.

[64] Anderson, M. R. D. Pose, and C. S. Wallace, *A Password-Capability System*, The Computer Journal, February 1986.

[65] Karger, P. A. and A. J. Herbert, *An Augmented Capability Architecture to Support Lattice Security and Traceability of Access*, Proceedings of the 1984 Symposium on Security and Privacy, IEEE, April 1984.

9. Acknowledgments

The research on Alpha and its constituent technology at Concurrent Computer Corp., SRI International, and Camegie Mellon University is sponsored by the U.S.A.F. Rome Laboratories, Computer Systems Branch. Additional support for Alpha is supplied by Digital Equipment Corp., and was provided at CMU by the U.S. Naval Ocean Systems Center, and the General Dynamics, IBM, and Sun Microsystems corporations.

The views contained in this paper are the authors' and should not be interpreted as representing those of the sponsoring organizations.

The authors are grateful to J. Duane Northcutt for his invaluable leadership of Alpha's first prototype design and development, and to Ed Burke, B. Dasarathy, Jim Hanko, Don Lindsay, Dave Maynard, Martin McKendry, Doug Ray, Sam Shipman, George Surka, Jack Test, Jeff Trull, Nick Vasilatos, Huay-Yong Wang, and Doug Wells for their essential contributions to Alpha. Teresa Lunt and Ira Greenburg at SRI International, together with Ray Clark at Concurrent Computer Corp. and Doug Wells at the Open Software Foundation, are the principals on the B3 multilevel secure version of the Alpha kernel. Alan Downing and Mike Davis of SRI International are the principals on the integrated resource management project for Alpha, assisted by Jon Peha at CMU.