

AD-A281 637



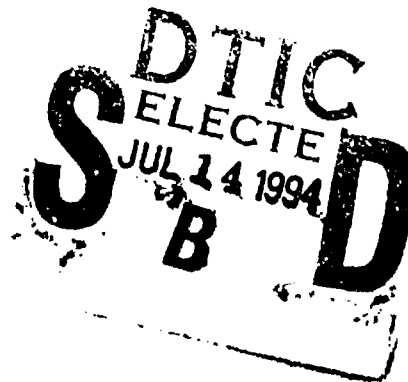
RL-TR-94-29
In-House Report
June 1994



1

THE SURVIVABLE DISTRIBUTED COMPUTING ENVIRONMENT

Patrick M. Hurley, Scott M. Huse



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

94-21570



128

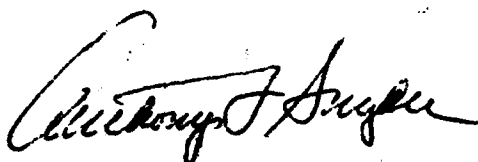
DTIC QUALITY INSPECTED 5

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-94-29 has been reviewed and is approved for publication.

APPROVED:



ANTHONY F. SNYDER, Chief
C2 Systems Division
Command, Control, and Communications Directorate

FOR THE COMMANDER:



JOHN A. GRANIERO
Chief Scientist
Command, Control, and Communications Directorate

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3AB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE June 1994		3. REPORT TYPE AND DATES COVERED In-House	
4. TITLE AND SUBTITLE THE SURVIVABLE DISTRIBUTED COMPUTING ENVIRONMENT				5. FUNDING NUMBERS PE - 62702F PR - 5581 TA - 28 WU - 17	
6. AUTHOR(S) Patrick M. Hurley, Scott M. Huse					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rome Laboratory (C3AB) 525 Brooks Road Griffiss AFB NY 13441-4505				8. PERFORMING ORGANIZATION REPORT NUMBER RL-TR-94-29	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3AB) 525 Brooks Road Griffiss AFB NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Patrick M. Hurley/C3AB (315) 330-2925					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) Distributed computer systems support several key attributes that are essential for the development and execution of command and control (C2) applications. Since C2 applications need to become more survivable, more dispersed, and better able to quickly adapt to new threats, we are seeking to provide an architecture for a survivable Distributed Computing Environment (SDCE). In essence, the SDCE will be a base upon which survivable distributed applications can be built. This base must be flexible enough to incorporate advances in technology. It must also be tailorable to the needs of specific C2 applications, and well structured for ease of maintenance. Hence, this base must be capable of evolving with the needs of C2 systems and their supporting technologies. The approach that was used in this effort was to utilize existing technologies such as the Mach micro kernel, along with the CRONUS and/or ISIS Distributed Computing Environments to provide many of the SDCE requirements.					
14. SUBJECT TERMS Distributed Operating Systems, Distributed System, ISIS, CRONUS, Mach, Chorus, Survivable, Distributed Computing Environment				15. NUMBER OF PAGES 20	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT U/L		

The Survivable Distributed Computing Environment

Patrick M. Hurley Scott M. Huse
Computer Systems Branch (C3AB)
Rome Laboratory
Griffiss AFB, New York 13441-5700

1.0 Introduction

Distributed computer systems support several key attributes that are essential for the development and execution of command and control (C2) applications. Rome Laboratory currently has a distributed computer systems in-house research and development laboratory called the *Distributed Systems Environment (DISE)*. The DISE is presently able to demonstrate and integrate many of these attributes. These include heterogeneity, replication, fault detection/recovery, and limited adaptive resource management.

In order for C2 applications to become more survivable, more dispersed, and better able to quickly adapt to new threats, we are seeking to provide and demonstrate a *Survivable Distributed Computing Environment (SDCE)* within the DISE. In essence, the SDCE will be a base upon which survivable distributed applications can be built. This base will be flexible enough to incorporate advances in technology. It will also be tailorable to the needs of specific C2 applications, and well structured for ease of maintenance. Hence, this base will be capable of evolving with the needs of C2 systems and their supporting technologies.

This paper is organized as follows. In section 2 the requirements for the SDCE are presented. Section 3 discusses the reasons for using micro-kernel technology as the base for the SDCE. Section 4 evaluates two candidate micro-kernel architectures Mach and Chorus. Section 5 defines the SDCE architecture. Section 6 presents other supporting technologies namely

Cronus and ISIS that are needed to fulfill many of the requirements of the SDCE. Section 7 takes a closer look at many of the attributes provided by Cronus and ISIS. It also presents an evaluation of Cronus and ISIS replication mechanisms. Replication is important because it is one of the best ways to detect and/or recover from faults in a distributed environment.

2.0 SDCE Requirements

The Survivable Distributed Computing Environment must provide the underlying mechanisms to support the development and execution of highly reliable distributed C2 applications. To accomplish this goal we list the requirements that should characterize the SDCE. Included in the list of requirements are distributed computing, fault tolerance, real time, adaptive resource management, trusted computing base, support for multi-domained applications, multi-clustered networks (interconnection of LANs and WANs), and heterogeneity (to include languages, operating systems and machine architectures). Note the SDCE may not readily provide all of these requirements. However, at a minimum the SDCE must provide the underlying mechanisms that are capable of supporting all the specified requirements.

Distributed computing may be defined as concurrent processes cooperating towards common goals where each process is likely to have incomplete and/or inconsistent global state information. As the name implies the SDCE should support distributed computing because it provides many desirable attributes. These attributes

include resource sharing (to include data), improved reliability, and increased performance (concurrent processing).

Fault tolerance may be defined as the ability of a system or component to perform its function, despite the presence of hardware or software faults. Fault tolerant mechanisms that detect and/or recover from hardware and software faults are essential for survivable systems. Therefore, the SDCE should also support underlying fault tolerant mechanisms in hardware, software, and communication. Numerous types of faults in hardware, software, and communication could be considered. However, due to the complexity of dealing with some faults, the requirements for the SDCE will be limited to the following candidate list.

Two classes of hardware faults should be supported by the SDCE - (1) the complete loss of a host/service; and (2) degradation in the performance of a host/service due to overload conditions. In terms of software faults, the SDCE should address - (1) algorithmic faults (perhaps through support of n-version programming); (2) software component faults (service loss); and (3) timing faults. Finally, two classes of communication faults should be considered - (1) loss of connectivity (local area network or wide area network); and (2) overloaded or congested communications.

The SDCE should also provide some support for real-time applications. Real time may be defined as a system or mode of operation in which computation is performed during the actual time that an external process occurs, so that the computation results can be used to control, monitor or respond in a timely manner to the external process. This would allow the SDCE to incorporate scheduling and resource decisions based on the current conditions of both system and environment within some specified time constraint.

Adaptive resource management may be defined as the ability of a system to change its internal state to be consistent with its external environment. Adaptive resource management should allow for both the static mapping of application components within the system as well as the ability to dynamically react to changes in the runtime requirements that could not possibly be anticipated. Therefore, a base for adaptive resource management should be supported to improve throughput and provide some level of fault avoidance. This base should also support both static and dynamic migration of processes and data.

The SDCE should also provide support for a trusted computing base. This trusted computing base should provide protection mechanisms within the computing environment to include hardware, firmware, and software, which cooperate to enforce security policies.

Support for multi-domained applications is also desirable. This would permit controlled access to resources (e.g., computers and data) in a flexible and efficient manner.

Typically network and distributed operating systems have provided support for these requirements through services such as processes, IPC, resource management, file server(s), etc. More recently, however, micro-kernel technology provides a new, more modular approach towards meeting these requirements.

3.0 Micro-kernels

Traditionally, network and distributed operating systems have been implemented by spreading knowledge about the system throughout large monolithic kernels. This monolithic design complicates the task of developing and integrating advances in technology with respect to both hardware and software. Fortunately, recent trends in

operating systems design have led to the use of micro-kernel architectures. This approach separates the components of the operating system that control hardware resources from those that determine the flavor of the operating system environment, e.g., a given file system interface. This allows the most complex software layers to be built on top of a relatively simple kernel (Figure 1).

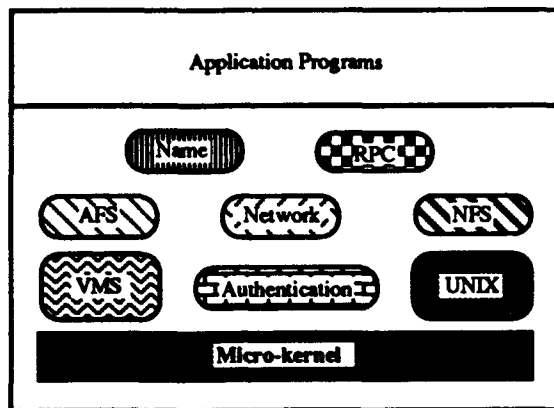


Figure 1. Micro-kernel Architecture.

Due to this modular layered, approach to operating system design, a micro-kernel architecture is able to offer a number of advantages over traditional network and distributed operating systems. Some of these advantages include : *tailorability* - the operating system environments can be customized for specific applications; *portability* - the operating system environment code is independent of a machine's instruction set, architecture, and configuration; *extensibility* - additional operating system environments and versions can be incorporated alongside existing systems; *real-time* - support for real-time applications is possible since the kernel is no longer required to hold long interrupt locks for UNIX system services; *multiprocessor support* - since the kernel is not required to support complex system functions (which may limit parallelism), greater parallelism is possible for its functions; furthermore, the micro-kernel's features can be better tailored to parallel

applications; *multicomputer support* - since the kernel only provides a relatively small number of basic abstractions, it can optimize the mapping of each abstraction onto the distributed hardware; and *security* - a smaller kernel is more easily defined and implemented in a secure manner; the modular layered architecture is simply better suited to trusted systems than that of traditional monolithic kernels [Black].

While the current level of maturity for micro-kernel technology does vary, some systems (e.g., Mach and Chorus) are already achieving commercially competitive levels of functionality and performance.

4.0 Mach and Chorus

This study evaluates two candidate micro-kernel architectures Mach [Black] and Chorus [Armand].

Some of the key features which each of these architecture's have in common include (1) a small kernel, (2) a modular architecture which provides scalability and allows dynamic configuration of the system and its applications, (3) transparent network access for interprocess communication, (4) a communication-based architecture which implements generic services used by a set of subsystem servers to extend standard operating system interfaces, (5) support for concurrency in both the operating system services and application programs; (6) support for large address spaces with flexible memory sharing, (7) integration of message passing communication with virtual memory, (8) real-time support that is accessible by system programmers, and (9) UNIX support.

4.1 Chorus

Chorus is an ongoing distributed systems research project conducted in France. To date, four versions have been developed. They are Chorus-V0 (1980-1982), Chorus-

V1 (1982-1984), Chorus-V2 (1984-1986), and Chorus-V3 (1987-present). Chorus-V3 was designed to integrate the best features of all the previous versions. Chorus was also designed with the intention of supporting industrial quality operating system environments.

The main abstractions implemented by Chorus include *actors*, *threads*, and *ports*. An actor is a collection of resources in a Chorus system. An actor defines a protected address space supporting the execution of threads that share the resources of the actor. A thread is the unit of execution in the Chorus system. A thread is a sequential flow of control, and, it is always tied to exactly one actor that defines the thread's execution environment. Within an actor, multiple threads can be created and can run concurrently. A port represents both an address to which messages can be sent and an ordered collection of unconsumed messages. When created, a port is attached to a specified actor. Only threads of this actor may receive messages on that port.

4.2 Mach

The Mach 3.0 micro-kernel architecture is being developed at Carnegie Mellon University. The history that led up to its development includes RIG (1976-1981), which led to Accent (1981-1986), which in turn was followed by the Mach 2.5 operating system (1986 - 1989). The Mach 3.0 micro-kernel (1989 - present) evolved from the Mach 2.5 operating system.

The basic abstractions of Mach are the *task*, *thread*, and *port*. A task may be viewed as a container to hold references to resources in the form of a port name space, a virtual address space, and a set of threads. A thread is an execution point of control. It is the basic computational entity. It belongs to one and only one task that defines its virtual address space. A port is a unidirectional communication channel between a client

who requests a service and a server who provides the service.

4.3 Chorus IPC VS Mach IPC

Mach and Chorus abstractions are very similar with respect to resource management (Chorus' actor and Mach's task), control (threads), and virtual memory. The addressing and communication abstractions of Mach and Chorus are, however, quite distinct.

In both Chorus and Mach, messages are addressed to intermediate entities called *ports*, not directly to threads or actors/tasks. It is the port abstraction that provides the necessary decoupling of the interface of a service and its implementation. This provides a basis for dynamic reconfiguration. Consequently, a port can migrate from one actor/task to another.

Addressing in Chorus is accomplished in a global manner via unique identifiers (UI). All Chorus objects (e.g., actors, ports) are referenced in this manner. The Chorus micro-kernel implements a UI location service that allows the referencing of Chorus objects without knowledge of their current location.

In contrast, Mach does not support the notion of global addressing, i.e., all ports' rights resolve to local ports. A network server extends Mach IPC across the network via the use of local *proxy* ports to represent remote ports. This collection of network servers (one on each node) then maintains the current location of network-wide ports.

The Chorus inter-process communication (IPC) mechanism permits threads to communicate through unreliable asynchronous point to *port group* (a port group is an abstraction that extends message passing semantics between threads by allowing messages to be directed to a

group of threads), or by synchronous reliable remote procedure call (RPC). When messages are sent to port groups, it is possible to: (1) broadcast to all ports in the group, (2) send to any one port in the group, (3) send to one port in the group located at a given site, and (4) send to one port in the group located on the same site as a given UI. Note, however, that the receive semantics are limited to one-to-one. That is, a port can only receive messages from a single sender (Figure 2a).

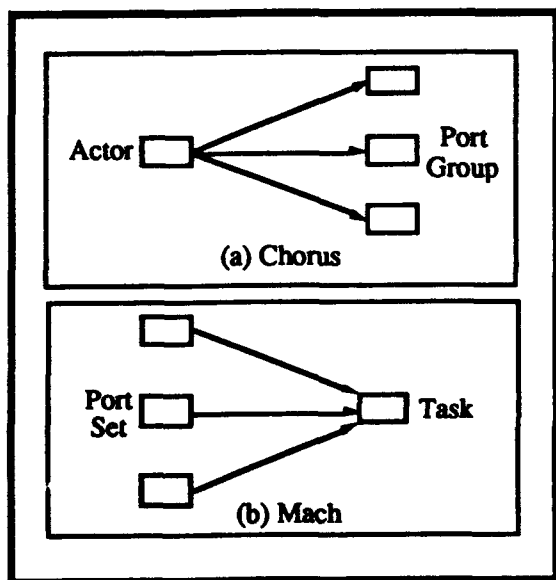


Figure 2. Send/Receive Semantics.

In contrast, Mach provides reliable point to point communication. With respect to the send semantics, the notion of a multicast is not present. The receive semantics, on the other hand, permit a task to receive messages from (potentially) multiple senders, known as a port set (Figure 2b).

4.4 Micro-kernel Selection

The SDCE micro-kernel technology base (along with other supporting technologies) must be capable of fulfilling all the SDCE requirements. This base must also be capable of evolving with the needs, and advancing technologies of C2 systems. Furthermore, since the micro-kernel

technology is the foundation for the SDCE, it must be a relatively mature and stable system.

Overall, the mechanisms provided by Chorus and Mach are comparable. The main difference lies in the IPC mechanism, as mentioned previously. Each of these IPC mechanisms have their strengths and weaknesses. Clearly, an ideal solution would be to combine the strengths of each of these abstractions. This very concept is nearing completion at Cornell University [Giade]. Work has also been done to improve the speed of Mach's IPC mechanisms [Draves] [Barrera III]. In addition, Real-Time Mach, and Distributed Trusted Mach are maturing along with the development of the Mach micro-kernel. Furthermore, it is clear that Mach is becoming an industry standard (e.g., Open Software Foundation). Due to these considerations we have decided to select the Mach micro-kernel for the foundation of the SDCE.

However, the Mach micro-kernel by itself does not meet all of the requirements of the SDCE. Therefore, we now present several different architecture design options that include supporting technologies (built on Mach or the Mach micro-kernel) which can meet the requirements of the SDCE. ISIS and Cronus are the specific supporting technologies. They provide support for distributed computing, fault tolerance, limited adaptive resource management, support for multi-domained applications, multi-clustered networks (interconnection of LANs and WANs), heterogeneity (to include languages, operating systems and machine architecture's), reliable communication, and concurrency. Note Cronus and ISIS will be discussed in greater detail later in this paper.

5.0 SDCE Architecture Options

The first architecture to be considered is built on the Mach 2.5 operating system

(Figure 3). This architecture is viewed as the easiest and lowest risk option because it is based on proven technology. However, it is not a micro-kernel based design and therefore it would be more difficult to adapt this system to the long term, ever-changing needs of C2 systems.

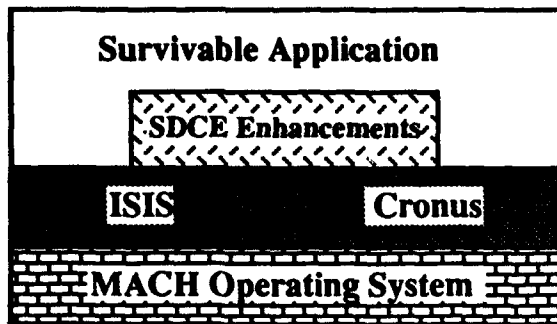


Figure 3. SDCE Architecture 1.

Unlike the architecture in Figure 3, the second architecture design option (Figure 4) utilizes the Mach micro-kernel. This architecture, however, requires an intermediate UNIX server to integrate the functionality provided by ISIS and Cronus. Although this architecture is micro-kernel based, it is not completely faithful to the micro-kernel design philosophy as illustrated in Figure 1. That is to say, ISIS and Cronus interact with the Mach micro-kernel through the UNIX server rather than directly with the Mach micro-kernel itself.

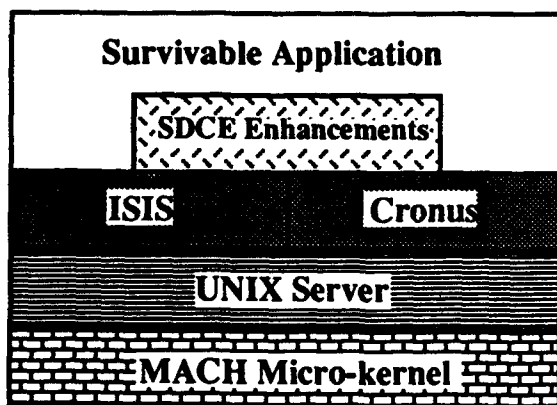


Figure 4. SDCE Architecture 2.

The third option (Figure 5) truly implements the micro-kernel design philosophy. Horus consists of an ISIS Toolkit [Birman] and IPC enhancements [Glade] to the Mach micro-kernel. Consequently, the need for an intermediate UNIX server is no longer required. This work is nearing completion at Cornell University.

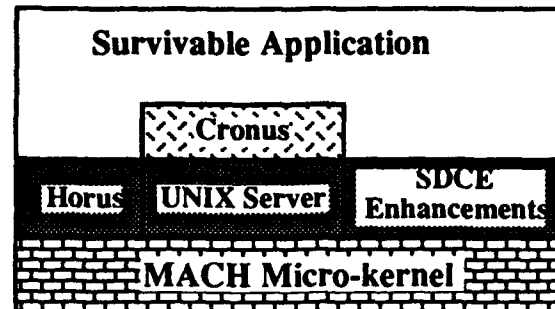


Figure 5. SDCE Architecture 3.

In addition, work is also being planned to build Cronus directly on top of the Mach micro-kernel. When available this capability could easily be incorporated into this architecture. In the interim, however, Cronus could rest on top of the UNIX server.

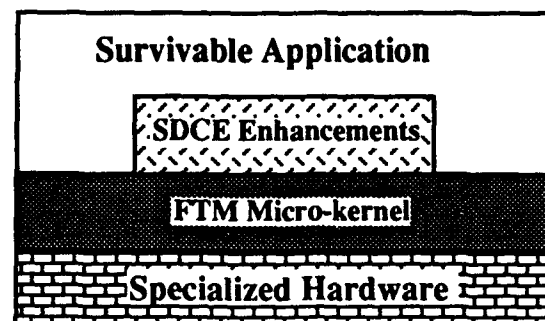


Figure 6. SDCE Architecture 4.

Figure 6 presents a hardware solution to the fault-tolerant requirements of the Survivable Distributed Computing Environment. The Fault Tolerant Multiprocessor (FTM) micro-kernel extends Mach by adding mechanisms that allow the implementation

of fault-tolerance. Normal Mach entities can be corrupted by a processor failure. The FTM architecture, however, can tolerate any single processor failure.

As previously mentioned, the architecture represented in Figure 5 is more consistent with the micro-kernel design, and provides the most versatile base upon which to build survivable distributed applications. It is therefore the architecture of choice, although the risk factor at this time may be relatively high due to the immaturity of Horus. In the event that Horus is too immature to be utilized within our time frame, the architecture represented in Figure 4 will be utilized instead. This does not preclude the incorporation of Horus at some future time. Note the SDCE enhancements in Figures 3 - 6 may include a reliable transaction service (e.g., Camelot), a replicated file server, or an X.500 server [Weider].

6.0 Supporting Technologies (Cronus and ISIS)

Cronus and ISIS are included in the SDCE architecture to provide support for distributed computing, fault tolerance, limited adaptive resource management, support for multi-domained applications, multi-clustered networks (interconnection of LANs and WANs), heterogeneity (to include languages, operating systems and machine architectures), reliable communication, and concurrency. Therefore a brief overview of these two systems will now be presented.

6.1 Cronus Overview

Cronus is a distributed computing environment that supports heterogeneous computer systems interconnected on a high-speed local area network (LAN) or wide area network (WAN). Cronus was funded by the U.S. Air Force (Rome Laboratory) and developed by BBN Laboratories Incorporated to support distributed

command and control applications. Present versions of Cronus provide support for such diverse systems as Sun workstations running Sun UNIX, DEC machines running VMS or ULTRIX, HP machines running HP-UX, and several parallel architectures. Cronus currently supports the following languages C, FORTRAN, and Common Lisp.

The Cronus distributed computing environment is based on the object model. An object consists of state information maintained in an object database and a collection of rules that govern how this state information may be examined or changed. Each rule represents an operation on the object. Objects and their associated operations are managed by object managers. Operations on objects are invoked by client programs or by other object managers.

Cronus object types define how the objects are to be used and implemented. Types are made up of operation code, operation interfaces, and data structures that specify the representation of the various objects. Types are also placed in a hierarchy structure that allows new types to be created as subtypes of existing ones.

Cronus consists of services, clients, and the Cronus kernel. Services (a service consists of one or more object managers) implement both system and application functions. Current system services provided by Cronus include an authentication service, a symbolic naming service (global), a network configuration service, a directory service, and an object type definition service. Clients within Cronus are processes that use services. The Cronus kernel itself sits on top of the native operating system and is primarily responsible for transmitting synchronous or asynchronous operation invocations from clients to services. The Cronus kernel makes the network appear transparent. For example, a client on one host in a given

Cronus configuration can invoke an operation on an object type whose manager resides on another host in the network (Figure 7).

By using an object-oriented approach, Cronus is able to provide a variety of higher level abstractions such as: (1) Management of replicated data (on disk); (2) Parallelism by splitting a computation among several machines; (3) Monitoring and reporting the status of a computation; (4) Dynamic reconfiguration from failures; (5) Support for multi-domained applications; and (6) Support for multi-clustered networks.

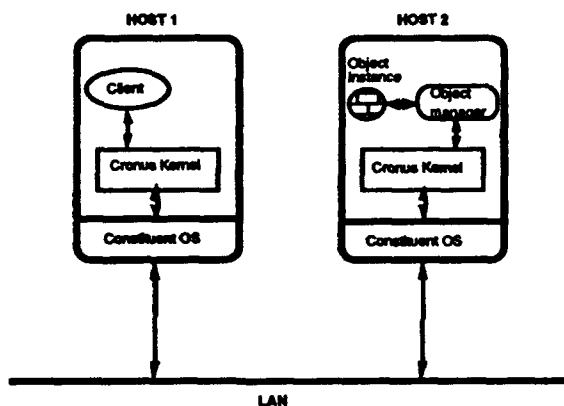


Figure 7. Cronus Communication.

6.2 ISIS Overview

ISIS is a distributed toolkit that provides high level tools which support the development of fast reliable distributed applications. ISIS provides mechanisms that support heterogeneous computer systems interconnected on a high-speed local area network (LAN) or wide area network (WAN). ISIS was funded by the Defense of Advanced Research Projects Agency (DARPA) and developed at Cornell University, Ithaca, New York.

The ISIS distributed toolkit is based on the concept of process groups. Process groups are a lightweight programming construct. A single process can belong to any number of groups and there is minimal overhead

associated with joining and leaving groups. Groups have a hierarchical namespace, much like a file system namespace, and permit flexible, location transparent addressing. Process groups are also capable of spanning across multiple machines. ISIS provides mechanisms for communicating atomically with a group, as one might do to inform its members of some event, or to issue them a request of some sort. Such a communication takes the form of a "multicast", in which one or all the members of the group receive the message, and zero or more respond (depending on the needs of the particular application).

Another major advancement that ISIS provides is the concept of virtual synchrony. Virtual synchrony permits the programmer to design a distributed program for execution in a simplified environment, wherein all processes observe events simultaneously and therefore in the same order. Events such as communication with a group or detection of failures are atomic in a virtual synchronous setting: all group members receive a message (or observe a failure) if any does.

Using process groups and virtual synchrony ISIS is able to provide a variety of higher level abstractions such as: (1) Management of replicated data (in memory or on disk); (2) Parallelism by splitting a computation among several machines; (3) Coordinate an external action; (4) Synchronize concurrent actions (such as when several processes share a resource that only one can use at a time); (5) Monitoring the status of a computation, process or computer, and triggering user-programmed defined actions should the status change; and (6) Dynamic reconfiguration from failures to include the integration of recovered machine into the operational system, restarting of the services that should run at that location and the ability to bring them up-to-date concerning the active state of the system.

The ISIS toolkit currently supports C, C++, FORTRAN, ADA and Common Lisp. ISIS runs on (and between) SUN, DEC, HP, GOULD, NEXT, and APOLLO equipment, and currently requires the UNIX or MACH operating system. However, ports to other operating systems such as AIX and VMS are being considered.

7.0 A Closer Look at Cronus and ISIS Mechanisms

It is now time to take a closer look at some of the mechanisms and/or attributes that Cronus and ISIS provide that help fulfill the requirements of the SDCE. As stated previously, Cronus and ISIS provide support for distributed computing, fault-tolerance, limited adaptive resource management, and support for multi-domained applications. We now break each of these capabilities up into the attributes and/or mechanisms they bring to the SDCE.

Distributed computing provides many desirable attributes, some of which are used as a base on which several other capabilities or attributes are built. These attributes or mechanisms include *Redundant Processing* - several processes performing the same task perhaps on different data; *Concurrent Processing* - work is divided among several processes, each of which will contribute to the final result; *Resource Sharing* - sharing data, memory, CPUs, disks, and other physical devices for a common goal, or because a host is deficient in the resources it requires; *Reliable Communication* - reliable communication between processes whether they are on the same host or on different hosts connected by LANs and WANs; and *heterogeneity* - the inter operability of languages, operating systems and machine architectures. Cronus and ISIS both support the attributes of distributed computing to some degree.

Fault-tolerant mechanisms that detect and/or recover from hardware and software faults are essential for a survivable system. Cronus and ISIS both provide and/or support a limited amount of fault tolerant mechanisms. These mechanisms and a brief description will now be presented.

Replication - Replication is defined as multiple copies of a resource maintained on different hosts to improve availability or consistency; *Triple Modular Redundancy (TMR)* - TMR is defined as three processors running the same code on the same data and voting on the result to mask out an error by any one processor; and *N-Version Programming* - N-Version Programming is defined as N versions of code (Using different algorithms) on the same data and voting on the results. The only one of these fault tolerant mechanisms that is directly supported by Cronus and ISIS is replication. The other mechanisms, however, can be supported but are more application specific.

Adaptive resource management mechanisms can be used to automatically adapt to evolving resource availability. Therefore, in the event of failures the system may automatically instantiate or migrate application functionality within and among the surviving system resources. ISIS provides some support for adaptive resource management that includes "recycling idle workstations for remote use and managing a pool of "compute servers". It can also manage a reliable replicated service by ensuring that some desired number of copies of the service are always running, despite machine failures. Cronus on the other hand provides no such mechanisms.

Mechanisms that provide support for multi-domained applications are also desirable. This would permit controlled access to resources (e.g., computers and data) in a flexible and efficient manner. Cronus supports multi-domained applications by a

mechanism called *clusters*. A cluster is a set of hosts grouped together into a single administrative unit. Each cluster is autonomous, and therefore, responsible for its own administration and control. No host is permitted to be a member of more than one cluster. Clusters allow boundaries to be erected between organizations, but can selectively allow or deny foreign clusters access to services that they support. This controlled access allows a cluster to provide remote access to only those services it desires. ISIS process groups, by default, provide a much less sophisticated mechanism (compared to Cronus) to support multi-domained applications. Process groups differ from clusters in that a member of one group can join any number of other groups. Process groups also have very little access control for joining a group, and once they become a member they can gain access to all the group services.

As previously stated C2 systems are inherently distributed and must be able to detect and/or recover from hardware and software faults. One of the best mechanisms in a distributed environment for detecting and/or recovering from faults is replication. Therefore, let's take a closer look at the replication mechanism provided by Cronus and ISIS. As you will see they take very different approaches to providing a replication mechanism.

7.1 Replication

Replication is defined as multiple copies of a resource maintained on different hosts to improve availability or consistency. Availability is achieved by relaxing the issues involved with keeping all copies consistent at all times. Consistency, on the other hand, is achieved by emphasizing the issues involved with keeping all copies consistent at all times. Therefore, a replication mechanism is capable of providing two different forms of survivability. Replication that stresses

availability is used for applications that require highly available data to function properly. Replication that stresses consistency is used for applications that maintain replicated data to survive faults.

7.2 Cronus Replication Overview

Cronus provides a replication mechanism that is based on the object model and version vectors, whereas ISIS provides a replication mechanism based on the concept of group programming and virtual synchrony. A brief overview of each of these approaches will now follow. This overview will then be followed by a more detailed comparison of these two very different replication mechanisms.

Cronus' replication mechanisms start by allowing the application programmer to select read and write quorums depending on his/her needs for data availability or consistency. To ensure maximum availability, the application programmer should select read and write quorums of one; this, however, will sacrifice consistency. To ensure data consistency the read quorum and write quorum should both be set to a majority ($N/2+1$), where N is the number of replicated copies. Version vectors are then set up for each replicated object. Each of these version vectors contain a list of hosts that support the same type of replicated object manager and an associated version number. These version vectors are updated each time the replicated object is accessed in order to reflect the current status of each replicated object. For every operation that is performed on the replicated object manager Cronus replication mechanisms collect the read or write quorums that were specified by the application programmer (note only the objects whose version vectors are up-to-date are considered) and locks those copies. The operation is then performed on one of the replicated objects, then that object is copied to all other accessible copies and the lock is released. If the read or write

quorum can not be obtained Cronus will return an error message to the application. If a previously inaccessible replicated object manager becomes available it is automatically brought into consistency using the version vectors.

In summary, Cronus supports replicated managers with consistency being maintained on a per operation basis by update then copy mechanisms. Also, by having a persistent data model as an integral part of Cronus managers, both process and persistent data replication are provided. Finally, the net effect of providing a replication mechanism this way is very similar to a transaction model.

7.3 ISIS Replication Overview

ISIS, on the other hand, provides no special tool for managing replicated data (in memory), because replication "falls out" directly when using process groups and virtual synchrony. This is because virtual synchrony guarantees that distributed events like broadcast message deliveries, notification of group membership changes (even if they are due to failures), and many other kinds of events will occur in exactly the same order in every process. A virtually synchronous system "looks synchronous" to every process in the system, but executes asynchronously when observed from the outside. More precisely, virtual synchrony guarantees the following (1) *Global event ordering* - All processes observe events in the same order; (2) *Causality* - An event E2 that is caused by an action occurring after some earlier event E1 will be observed everywhere after E1; and (3) *Atomicity* - Event notification is all-or-nothing.

ISIS replication starts by an application programmer creating a process group that manages the same data (in memory). This process group must provide state transfer routines to ensure that new member(s) to the group receive the current state when

they join the group. This form of replication is called *process replication* because all data will be lost if there is a simultaneous failure involving all the processes maintaining the replicated data. If the state of the data must be maintained despite this type of failure then persistent data is required. Persistent data is maintained in ISIS by saving state to stable storage using a logging mechanism. This logging mechanism periodically logs a copy (a checkpoint) of the group state onto stable storage logging all changes that occur to the state. The frequency of the check point can be controlled by the application programmer. Thus the desired degree of consistency can be achieved.

In summary, ISIS supports replicated service via process groups and virtual synchrony. Unlike Cronus, read and write quorums are dynamic and based on the current group membership and the particular client/server communication semantics (may be different for different clients). Consistency is managed via ISIS communication protocols that support the model of virtual synchrony. Virtual synchrony is based on the transaction model but permits a "looser" coupling between servers (more asynchrony) which provides better concurrency.

7.4 Cronus vs ISIS Replication Evaluation

A test application was built utilizing a *client-server* model. In this model clients make calls to a population of servers which can be resident on several nodes in the system. The servers are passive entities waiting for an invocation from a remote client.

Care had to be used to insure fairness when evaluating these two very different replication mechanisms. This was accomplished by guaranteeing that the two systems provided exactly the same service. This service required the ability to create, delete and update data in stable storage.

Note stable storage is important because it allows the application to recover from a total failure.

In our test application the server is responsible for the implementation of and execution of creating, deleting and updating data in stable storage; whereas the client is only responsible for invoking the create, delete and update operations on the server(s). The general flow of processing in our test application is described as follows: (1) The client obtains the local system time to record the start time of the experiment; (2) The client then invokes "N" operations on the server. Where "N" can range from 1 - 1000 and is used to obtain the average time it takes to complete the specified operation; (3) The replicated server then executes the operation; (4) The client waits for a response from the server that the operation completed successfully; (5) the client obtains the local system time to record the finishing time of the experiment.

These steps are executed for the create, delete and update operations to determine the average time each operation takes. The number of servers used to perform the creates, deletes, and updates is also varied from 1 - 3 to measure the time each operation takes for different degrees of replication. This application is executed by both Cronus and ISIS to determine what replication mechanism provides a better service under what conditions. Note that both test applications (Cronus and ISIS) were set up to maximize consistency. The results of this evaluation will now be presented.

The software configuration consisted of the Sun OS version 4.1 as the constituent operating system. Layered on top of this operating system is Cronus 3.0 and ISIS 3.0.8 which are the distributed environments under evaluation. The hardware configuration for the evaluation consisted of three Sun SPARC 1+

workstations with 16 MB of memory. These workstations were connected on a local area network with a bandwidth of 10 MB/sec and were using IEEE 802.3 media access protocol. The whole system was dedicated to this experiment with no other load on it.

Although the Sun OS was used in this evaluation instead of the Mach micro-kernel, as described by the SDCE architecture, we believe that the conclusions presented in this section are valid when comparing the supporting technologies. We do, however, anticipate proving this claim in the near future by repeating the replication evaluation on Cronus and ISIS supported by the Mach micro-kernel.

Replication Evaluation Results in seconds						
Number of copies	Cronus			ISIS		
	Create	Delete	Update	Create	Delete	Update
1	.0424	.0200	.0389	.0126	.0126	.0122
2	.0439	.0208	.1066	.0178	.0177	.0180
3	.0442	.0224	.2536	.0220	.0218	.0221

Figure 8. Replication Evaluation

The results of this evaluation are shown in Figure 8. In general the data produced from this evaluation did not provide results that were too surprising. For example we notice that the overhead increases incrementally as we increased the number of replicated copies. The only really surprising observation was in the Cronus "update" results. The "update" operation appeared to perform poorly for two reasons (1) Every object in the object database must be pulled out of stable storage to find the item that is to be updated. (2) Also the "update" operation makes another operation invocation to "finditem" which is responsible for finding the item on which to perform the update. This adds additional

overhead to the "update" operation because the "finditem" operation now has to worry about the issues involved with maintaining consistency, such as voting to obtain a read quorum. ISIS does not suffer from these problems in the "update" operation because (1) The data does not have to be pulled out of stable storage to search for the item to be updated. This is because the data is maintained in memory and the only time stable storage is used is when operations are invoked that change the state of the data; and (2) Although the ISIS version of the "update" operation also invokes the "finditem" operation, ISIS does not have to worry about voting mechanisms to ensure consistency because virtual synchrony automatically guarantees consistency.

The replication mechanisms within ISIS are more flexible and dynamic than those found within Cronus. For example ISIS requires no pre-defined and pre-compiled read or write quorum as does Cronus. However, Cronus replication mechanisms are better able to maintain consistency during and after a network partition. This is because Cronus does have pre-defined read and write quorums. Cronus allows a network partition with a majority of the copies to continue execution and upon recovery from the network partition guarantees that all copies are brought to a consistent state. ISIS, on the other hand, provides a mechanism where group members (replicated copies) can be "weighted" by the application designer so in the event of a network partition the side with the largest "weight" continues as the primary partition, all the other partitions continue as secondary partitions. Note all partitions know if they are in a primary or secondary role. Upon recovery there may be inconsistencies that require human intervention to resolve. However, the application designer could guarantee that no changes to stable storage occur in a secondary partition thus maintaining consistency.

In summary, ISIS appears to provide a better replication mechanism than Cronus provides. However, the SDCE has many other requirements and much more work has to be accomplished before a determination can be made in regards to those requirements.

8.0 Acknowledgments

The authors of this paper would like to acknowledge Dr. Gary L. Craig (Syracuse University) for his useful discussions in micro-kernel technology. We would also like to thank him for his comments in reviewing this paper.

9.0 References

Armand, F., et. al., Towards a Distributed UNIX System - The Chorus Approach, *Proceedings of the European UNIX Systems User Group Conference*, September 1986.

Barrera, Joseph S. III, A Fast Mach Network IPC Implementation, *Usenix Association*,

Birman, Kenneth P., The Process Group Approach to Reliable Distributed Computing, Department of Computer Science, Cornell University, July 1991.

Black, David L., et. al., Micro-kernel Operating System Architecture and Mach, *Usenix Association*, pp. 11-13.

Cooper, Robert C. B., Glade, Bradford B., Birman, Kenneth P., and Robert van Renesse, Light-Weight Process Groups, Department of Computer Science, Cornell University, 1992.

Draves, Richard, A Revised IPC Interface, *Usenix Association*.

Glade, Bradford B., Birman, Kenneth P., and Robert van Renesse, Group Communication in Mach: Kernel Interface Supplement, Department of Computer

Science, Cornell University, November 3,
1992.

Weider, Teynolds, and Heker, Technical
Overview of Directory Services Using the
X.500 Protocol, NIC RFC 1309, 16 pages,
March 1992.

***MISSION
OF
ROME LABORATORY***

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.