

**DTIC FILE COPY**

4

**RADC-TR-88-82**  
**Final Technical Report**  
**April 1988**



**AD-A199 721**

# **CRONUS, A DISTRIBUTED OPERATING SYSTEM: PHASE I**

**BBN Laboratories Incorporated**

**Richard E. Schantz, Robert H. Thomas, R. Gurwitz, G. Bono, M. Dean,  
K. Lebowitz, K. Schroder, M. Barrow, and R. Sands**

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

**DTIC**  
**ELECTE**  
**OCT 04 1988**  
**S H D**

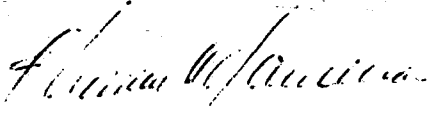
**ROME AIR DEVELOPMENT CENTER**  
**Air Force Systems Command**  
**Griffiss Air Force Base, NY 13441-5700**

**88 10 3 106**

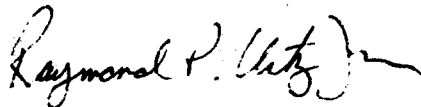
This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-82 has been reviewed and is approved for publication.

APPROVED:

  
THOMAS F. LAWRENCE  
Project Engineer

APPROVED:

  
RAYMOND P. URTZ, JR.  
Technical Director  
Directorate of Command & Control

FOR THE COMMANDER:

  
JOHN A. RITZ  
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b RESTRICTIVE MARKINGS N/A		
2a SECURITY CLASSIFICATION AUTHORITY N/A			3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4 PERFORMING ORGANIZATION REPORT NUMBER(S) 5885			5 MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-82		
6a NAME OF PERFORMING ORGANIZATION BBN Laboratories Incorporated		6b OFFICE SYMBOL (if applicable)	7a NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)		
6c ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge MA 02238			7b ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		
8a NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center		8b OFFICE SYMBOL (if applicable) COTD	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0132		
8c ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			10 SOURCE OF FUNDING NUMBERS		
		PROGRAM ELEMENT NO 63728F	PROJECT NO 2530	TASK NO 01	WORK UNIT ACCESSION NO 07
11 TITLE (Include Security Classification) CRONUS, A DISTRIBUTED OPERATING SYSTEM: PHASE I					
12 PERSONAL AUTHOR(S) Richard E. Schantz, Robert H. Thomas, R. Gurwitz, G. Bono, M. Dean, K. Lebowitz, K. Schroder, M. Barrow, and R. Sands					
13a TYPE OF REPORT Final		13b TIME COVERED FROM Sep 81 to Sep 84	14 DATE OF REPORT (Year, Month, Day) April 1988		15 PAGE COUNT 84
16 SUPPLEMENTARY NOTATION N/A					
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
12	07		Distributed Operating System Interoperability Heterogeneous Distributed System Survivable Application System Monitoring and Control		
19 ABSTRACT (Continue on reverse if necessary and identify by block number)  This is the final report for the first phase of development for the CRONUS DOS Design and Implementation Project. CRONUS is the name given to the distributed operating system (DOS) and distributed system architecture application development environment being designed and implemented by BBN Laboratories for the Air Force Rome Air Development Center.					
20 DISTRIBUTION/AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a NAME OF RESPONSIBLE INDIVIDUAL Thomas F. Lawrence		22b TELEPHONE (Include Area Code) (315) 330-2158		22c OFFICE SYMBOL RADC (COTD)	

DD Form 1473, JUN 86

Previous editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

## Table of Contents

1	Introduction.....	1
1.1	The Role of a DOS.....	2
2	Overview.....	5
2.1	Background.....	5
2.2	Functional Description.....	5
2.3	System Design.....	6
2.4	System Implementation.....	8
2.5	Test and Evaluation.....	9
3	The Cronus Architecture and Design.....	10
3.1	General Hardware Architecture.....	10
3.2	Key Ideas and Implications.....	12
3.3	Software Architecture.....	14
3.4	The Cronus Object Model.....	16
3.4.1	Cronus Objects and Operations.....	16
3.4.2	Layered Structure.....	18
3.4.3	Message Passing Core.....	21
3.4.4	Primitive Operations.....	22
3.4.5	Object Location and Message Routing.....	23
3.4.6	Properties of Messages.....	24
3.4.7	Message Encodement.....	25
3.4.8	Extending Beyond a Local Network.....	27
3.5	Basic Cronus Types.....	29
3.6	Access Control in Cronus.....	30
3.7	File System.....	32
3.7.1	Primal Files.....	34
3.7.2	Reliable Files.....	37
3.8	The Cronus Catalog.....	44
3.8.1	Implementation of the Cronus Symbolic Catalog.....	47
3.9	Automating Cronus Manager Development.....	51
3.9.1	Manager Facilities Provided Automatically.....	51
3.9.1.1	Multiple Object Types.....	52
3.9.1.2	Dispatching.....	52
3.9.1.3	Multitasking.....	53
3.9.1.4	Access Control.....	53
3.9.1.5	Inheritance of Operations.....	53
3.9.1.6	Message Parsing and Validation.....	54
3.9.1.7	Storage For Instances of Objects.....	54
3.9.2	Client Facilities Provided Automatically.....	55

3.9.2.1	Subroutine Interfaces	55
3.9.2.2	Generic User Interfaces	55
3.9.3	Documentation	55
3.9.4	Experience to Date	56
3.10	Cronus Monitoring and Control System	56
3.10.1	Role of the MCS	56
3.10.2	Functional Areas	57
3.10.2.1	Fault Detection	57
3.10.2.2	Logging	58
3.10.2.3	Fault Isolation	58
3.10.2.4	Fault Correction	59
3.10.2.5	Resource Allocation and Policy Management	59
3.10.3	Current Implementation	60
3.10.3.1	Host Probes and Service Probes	60
3.10.3.2	Transaction Log	61
3.10.3.3	Status Display Programs	61
3.10.3.4	Starting and Stopping Services	61
3.10.3.5	Graphical User Interface	61
3.10.3.5.1	Graphical Presentation	62
3.10.3.5.2	Interactive	62
3.10.3.6	Configuration Management	63
3.10.3.7	Structure of the MCS	64
3.11	Resource Management in Cronus	64
3.11.1	General Approach	64
3.11.2	A Resource Management Example	67
4	Test and Evaluation	71
4.1	Areas of Internal Use	71



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	

## 1 Introduction

This is the final report for the first phase of development for the Cronus DOS Design and Implementation Project. Cronus is the name given to the distributed operating system (DOS) and distributed system architecture application development environment being designed and implemented by BBN Laboratories for the Air Force Rome Air Development Center.

This is not the first DOS developed by BBN. The field of distributed operating systems and its supporting technology base has evolved rapidly since our early attempts over the preceding ten years at coupling the operation of otherwise autonomous systems. In this respect Cronus represents an amalgamation of much of what we have learned from previous DOS efforts combined with the experiences of others pursuing similar goals, new underlying technology, and updated views of what distributed operating systems and programming distributed systems are all about.

The Cronus distributed operating system is intended to promote resource sharing among interconnected computer systems, and manage the collection of resources which are shared. Its major purpose is to provide a coherent and integrated system based on clusters of interconnected heterogeneous computers to support the development and use of distributed applications. Distributed applications range from simple programs that merely require convenient reference to remote data, to collections of complex subsystems tailored to take advantage of a distributed architecture. One of the main contributions of Cronus is a unifying architecture and model for developing these distributed applications, as well as support for a number of system provided functions which are common to many applications. The Cronus project is starting from a desire to achieve the basic concept of an integrated distributed computer utility and evolving a design and implementation toward experimentally validating its system model.

## 1.1 The Role of a DOS

It is now a practical reality that the world is and will continue to be populated by distributed computing resources. This is a direct outgrowth of two somewhat distinct but related phenomena. The first factor is the increase in the number of computerized activities associated with any given organization of significant size along with the inherent distribution within the organization itself. The second factor is the tremendous decrease in the price and size of computer systems and modules making it economically feasible for a system designer to contemplate a collection of intelligent computer systems (components) as a solution to some data processing requirements.

As a result of the first factor, organizations often find themselves with a number of independent, sometimes unique computer systems, each supporting some of their overall data processing requirements. When this situation arises, it is quite natural to seek degrees of interoperability between the previously independent computer systems by interconnecting them via a computer network. One important role of a DOS is to support and control interoperability between at least some of the resources developed by the constituent host systems. The second factor (the feasibility of using collections of machines to provide some integrated service) leads to a quite a bit different view of the role of a DOS. The contribution of any single host toward supporting the overall system functionality may be limited. A number of potential benefits are often cited for organizing a set of services around a collection of cooperating systems. Among these are the potential for increased efficiency through functional specialization, modular expansion, forms of enhanced reliability, and potentially improved performance through extended parallelism.

Applications which could make use of distributed resources come in a variety of forms, each often attempting to initially utilize their distribution as a means of achieving a different goal. It is believed by many that a distributed architecture is key to increasing the reliability, or improving the performance or modular expandability of computing systems. Distribution is key to system reliability because physical distribution (even over relatively short distances) typically means independent failure patterns. It is key to system performance because of potential for increased efficiency through functional specialization and because distributed processing components can exhibit a great deal of real parallelism. It is key to system

modularity and evolvability because of the requirement for well defined functional interfaces and protocols and the convenience of adding additional computing elements.

There are other approaches to reliability, performance and modularity besides a distributed one. In isolation other approaches could even be as promising. However, when taken all together, and included with such other desirable features such as interoperability between heterogeneous system resources, and shared access to common services, a distributed processing computer utility appears to be the only approach capable of simultaneously achieving many or all of these objectives. Distributed processing systems represent not so much a difference in terms of what is computed but rather in how a system is structured to provide the computational facilities.

Despite the growing existence of distributed resources, it is decidedly not a reality that these resources can be easily integrated into a coherent collection responsive to the needs of the organization deploying them. Bits and pieces of extremely valuable functionality applying distributed resources to emerging immediate problems continue to be developed, but there have been few attempts to try to either capture the basis of distributed applications to make it available for subsequent ones, or to develop an application in such a way that it can be easily integrated with preceding, concurrent or planned applications in an integrated fashion as part of the overall automation objectives of the organization. It is the distributed operating system and its associated hardware, firmware and software which organizes an otherwise independent collection of computers into an effective computing facility.

The two distinct aspects of a DOS, for preplanned and unplanned integration, are reflected in the various parts of our DOS design. Some parts are oriented toward the interoperability of similar functions found on many individual systems, thereby providing a degree of uniformity in an otherwise heterogeneous environment. Other parts are more concerned with value added to individual functions within an environment which is completely designed to operate with companion functions in a more homogeneous fashion.



It is the intent of the Cronus design and testbed implementation to provide a hands-on vehicle for addressing the technical and administrative issues associated with trying to effectively utilize distributed system technology.

In the rest of this report we discuss the background for the project and how it is organized. We then provide an overview of the architecture and describe the system design and implementation to date. We conclude with a synopsis of our test and evaluation experiences to date.

## 2 Overview

### 2.1 Background

Recognizing the potential that a distributed systems architecture has for meeting future command and control application requirements for interoperability, survivability and extensibility RADC has been supporting the development of applicable technologies through its Distributed System Technology Program. One of the areas being pursued is that of distributed operating systems. A distributed operating system (DOS) is that software which integrates, coordinates and controls a collection of computer systems interconnected by means of one or more communication networks. A DOS serves to facilitate development and operation of application programs requiring the resources of multiple machines for reasons of functionality, survivability, scalability, and/or performance.

In 1981, after a number of study projects, and experience with a number of prototype distributed system implementations, BBN began work on developing for RADC a distributed system testbed and DOS software. This distributed system testbed is intended to provide a base for building and evaluating distributed applications. The DOS Design/Implementation project was divided into four parts: functional description, system design, system implementation, and test and evaluation.

### 2.2 Functional Description

The first part of the Cronus project involved assessing the requirements for use of distributed system technology in supporting future C2 systems. This resulted in a Cronus System Functional Description which lays the framework for the system development. Two prominent themes, which are similar to the development of the C2 systems themselves, emerged from the functional description effort. The first is that the problem area is multi-dimensional and potentially very large, and the second is that the systems developed must evolve in such a way that attention can be focused on specific parts of the overall problem at different times. The functional description enumerated the various functional areas and system properties which were thought necessary for successful demonstration of distributed applications.

The system objectives were to establish a comprehensive distributed system architecture, and then to integrate a collection of computer systems into a coherent and uniform computing facility within this architecture. This computing facility should exhibit the following properties:

- o survivability of system functions
- o scalability of system resources
- o global management of system resources
- o substitutability of system components
- o convenient operation of the system

Our approach to developing Cronus has been to establish a general system framework into which each of the above-mentioned objectives fit, and then to elaborate in these areas on a priority basis. The Cronus effort to date has concentrated on developing an extensible distributed system architecture, establishing an initial Cronus hardware testbed facility, designing and implementing a model for host-independent access to system resources, establishing system-wide uniformity in a variety of DOS functional areas, and begun to address issues of survivability, resource management, and monitoring and control.

The functional areas which were determined to be necessary include flexible interprocess communication, a system-wide global name facility, authentication and access control, a distributed file system, distributed process management, a uniform user interface, and monitoring and control software.

### 2.3 System Design

The second part of the effort involved the design of a system meeting the above functional requirements. Because of the experimental nature of the effort, a system design approach which supported system evolution was adopted. This design allowed a multi-phase implementation effort, where some parts of the system could be designed and implemented before other parts. The system design was documented in a series of Cronus System/Subsystem reports. These reports describe the overall system structure,

established the major components of the system, and developed designs for those parts of the system which were to be constructed during the initial phase of implementation. The intent of our initial design work was to establish a solid framework which could be extended in an orderly way in all of the areas of interest, focusing on demonstrating particular aspects of distributed system technology.

The Cronus system design is based on an object-oriented view of the system. The Cronus system kernel includes a host transparent object-based interprocess communication facility, focusing on the invocation of operations on objects. The kernel itself supports the abstractions of a host object for monitoring and control purposes, and process objects for supporting Cronus object managers. A Cronus library provides a standardized interface for invoking operations on objects, including conversion to and from a standard data exchange format for inter-processor communication (thus accounting for heterogeneity within the cluster). The rest of the system consists of system and application objects and managers, along with client commands and subsystems which allow users to access these objects and perform operations on them. Some of the system objects are migratable, and serve as a basis for reconfiguring the system, while other objects are replicated to support survivability. The initial set of object managers used as building blocks supporting Cronus application software include:

- o Catalog managers, which collectively implement a system-wide symbolic name space;
- o authentication managers, which provide principal and group objects supporting authentication and access control,
- o file managers, which implement a distributed file system,
- o device managers, which allow connection of terminals, line printers, etc.

Initial application software provides the user interface and monitoring and control functions. The system design is extensible through the addition of new object types, including application-specific objects. In this way, the development of system functions (such as a catalog manager) serves both as an example of how to use system mechanisms for supporting application development, and as a building block for new applications.

## 2.4 System Implementation

The third aspect of the Cronus project is the development and implementation of a testbed facility. This includes the selection and integration of a variety of hardware components representative of the diverse selection expected to be found in a command and control data processing facility. It also involves the implementation of the Cronus design for these hardware components.

The initial hardware components specified and selected for integration into the Cronus testbed were C/70 UNIX and VAX/VMS as general-purpose application hosts (demonstrating processor and operating system heterogeneity) and special-purpose microprocessor-based Generic Computing Elements (GCEs), interconnected via an Ethernet local area network for communication within a cluster, and standard Internet gateways for communication between clusters. In all cases, these components represent either an instance of a larger class of hardware types, or a demonstration vehicle (with possible later substitution of a component more appropriate to a given operational context). An additional system component, a dedicated single user workstation, was also included in the original architecture. Because of the current volatility of workstation technology, workstation integration was deferred from immediate attention. However, we have recently selected and integrated a SUN microsystem workstation to serve the role as a dedicated C2 workstation. A recent system upgrade also included evolving the C/70 based UNIX software onto 4.2BSD VAX-UNIX systems.

To ensure the substitutability of the network communication medium, the system implementation is based on the standard DoD protocols, IP and TCP.

Both the Cronus design and implementation are proceeding in phases. Our approach to implementation is first to concentrate on an initial version of basic system functionality, demonstrating the uniformity, coherence, and flexibility of the system concept within the heterogeneous cluster environment. Although the general approach to system survivability and cluster-wide resource management in Cronus has already been defined and some initial demonstration software is in place, developments in these areas are the dominant functional extensions specified as part of the next phase of implementation.

To date we have implemented the Cronus object system support software for each host in our testbed configuration. In addition, we have implemented a variety of object managers which represent major system functional units. These include catalog managers, file managers, and authentication managers all running on many of the host types in the configuration and integrated with initial application programs. An initial monitoring and control capability has also been developed.

## 2.5 Test and Evaluation

The fourth aspect of the Cronus project is system test and evaluation. The general approach taken is to promote the use of Cronus system components to support the daily activities of the Cronus developers and further development of the system. This focus was intended to accomplish multiple objectives. First, it causes the use of system components which shake out reliability and performance problems early in the development cycle. Second, it concentrates effort on the problems of developing software for a Cronus cluster. Since a major role of emerging distributed operating systems is support for the development of distributed applications, this form of test and evaluation helps in smoothing the transition from in-house usability to use by outside organizations. Use of the system by the system developers has meant early emphasis on the distributed file system aspects of Cronus, as well as attention to the problems of integrating existing software development tools into the Cronus environment. This too is an important area for continued implementation effort.

### 3 The Cronus Architecture and Design

#### 3.1 General Hardware Architecture

Cronus operates in an environment made up of interconnected computer communication networks. This internet environment includes both geographically distributed networks which span tens to thousands of miles and local area networks which span distances of up to a mile or two. From an architectural point of view, it is useful to think of this environment as being composed of clusters of host computers, where the hosts within a cluster are separated by distances of up to a few miles and are typically under a unified administration.

A cluster is specified by an explicit enumeration of its host components. The configuration of a cluster may change over time by the addition and removal of hosts. These changes are expected to occur relatively slowly. Since a cluster, in effect, acts to define the boundaries of a Cronus system, a cluster is the domain over which a class of names are guaranteed to have meaning and be interpretable. The names of interest here are both low level unique identifiers and higher level symbolic names for objects.

A cluster may include hosts on several networks, and several clusters may exist on the same network. However, performance considerations will generally lead to clusters that consist of hosts on a single local area network or on a few local networks interconnected by means of high performance gateways. Therefore, although a cluster is a logical rather than a physical concept, clusters will tend to be aligned with local area networks.

Cronus currently operates in a cluster defined by one or more local area networks. Extensions to multi-cluster architectures are currently being designed. The principal elements in a Cronus cluster include:

1. A set of hosts upon which Cronus operates.
2. One or more high performance local area networks which support communication among hosts within a Cronus cluster.
3. An internet gateway which makes the cluster part of the internet by supporting communication between cluster hosts and hosts external to the cluster.

The Cronus host set is a heterogeneous collection of hosts which can be divided according to function into three broad classes:

1. Hosts dedicated to providing Cronus functions.

The functions the hosts provide include file and data storage, user authentication, catalog management, device control and terminal access. The hosts which support these functions are called Generic Computing Elements (GCEs).

GCEs are small, dedicated-function computers of a single architecture but varying configuration. Each GCE provides one or more basic Cronus functions. Since GCEs have the same architecture, they provide a replicated hardware resource which, with appropriate software, enhances the survivability of basic Cronus functions.

2. Utility and application hosts.

Although these hosts may support some Cronus functions supported by GCEs, their primary role is to support user applications. The utility and application hosts include a variety of machines with unrelated architecture. They are typically mainframe hosts which may serve a number of users simultaneously.

These hosts run operating systems which are largely unmodified. The software necessary to integrate them into Cronus runs as an adjunct to rather than a replacement for the hosts' primary operating systems. Hosts can be included in Cronus with varying degrees of system integration, with some supporting and providing access to only limited subsets of the services defined by the Cronus environment.

3. Single user workstations.

Workstations are powerful, dedicated computers which provide substantial computing power and graphics capability to a single user. They are used both to provide user access to Cronus and for their ability to run applications. They differ from mainframes in that they support a single user and from terminals in that they offer significant computational resources.

Since we began the Cronus design three years ago, a new host type has emerged: the inexpensive personal computer as exemplified by the Apple Macintosh or the IBM PC. We are currently considering



the architecture impact, if any, of these single user machines.

### 3.2 Key Ideas and Implications

There are several key ideas which form the basis for the Cronus design. These are:

1. The Cronus object model.

The basic system organizing principle for Cronus is an abstract object model. Cronus can be thought of as a collection of objects organized into classes called types. The services Cronus supports are implemented by processes that manage various object types: files, processes, directories, etc. Cronus attempts to treat all types uniformly, in accord with its object model. Within the object model, location independence and dynamic run time binding to objects are important concepts behind the design. The object model is extendable to application development. The system design is based on the idea that to a user there is essentially no difference between providing "system" services and "application" services.

2. System-wide availability of essential services.

The services provided by Cronus include:

- o Object management for many types of Cronus objects.
- o A standard interprocess communication (IPC) facility.
- o A system-wide distributed file system.
- o A system-wide symbolic name space for all types of objects.
- o Facilities for process management.
- o User and process authentication.
- o A standard access control discipline for all system and application resources.
- o A user interface that provides access to all Cronus and application services.

- o Monitoring and control services for the entire system, the individual hosts, individual services, and applications.

At the heart of the Cronus concept is the availability of these services to all Cronus applications. The coherence and uniformity of Cronus is directly enhanced when applications and application host operating systems utilize the Cronus-supplied services as the single source of these services. To the extent that applications and application hosts choose to utilize parallel but incompatible services, coherence and uniformity are diminished. For many existing applications and hosts integrated into the Cronus architecture, we anticipate a gradual evolution from dependence on local mechanisms to reliance on Cronus functions for similar globally managed services.

- 3. Generic Computing Elements (GCEs).

The concept of the GCE is important to the Cronus design. As previously noted, a GCE is an inexpensive host that can be flexibly configured with small or large memory, and with or without disks and other peripherals. GCEs are configured for, and dedicated to specific Cronus services, such as file storage, Cronus catalog management, and user authentication. Because they are dedicated to Cronus, it is possible to control and optimize the performance and reliability of the Cronus services supported on GCEs.

- 4. Minimal dependence on a particular LAN technology and other system hardware.

Cronus accesses the local network capabilities indirectly through an interface called the Virtual Local Network (VLN) rather than directly. The VLN interface embodies an abstraction of local network capabilities. The cluster configuration used for Cronus development includes an Ethernet. By building Cronus to use the VLN interface, it is possible to replace the Ethernet with any local network that provides the basic transport services Cronus requires simply by developing software which implements the VLN interface for the new network. Cronus has already been ported to a Pronet Ring network base with only network device drivers needing recoding. Writing Cronus system software in a high level language (C) and using Cronus mechanisms internal to the implementation both reduce dependencies on constituent operating system functions. This makes Cronus components easy to port to new hardware which is anticipated to increase the effective system

lifetime through a number of periods of hardware evolution.

5. Flexible host integration.

When a new application host is integrated into Cronus there are a range of integration possibilities which occupy different points in a cost versus degree of integration space.

When a host is integrated with minimal effort, little more than a communication path between the host and the rest of Cronus will be present. The host will be able to obtain many Cronus services through the communication path, but its own resources may be inaccessible to external process through the normal Cronus mechanisms. Further effort can be devoted to integrate the host more fully into Cronus.

6. Use of Standards

Cronus uses recognized standards in several key areas. These directly contribute to both the coherence of Cronus and interoperability with other systems. Standards are used predominantly as interfaces internal to the system itself, as a means of simplifying the introduction of new system components which already adhere to one or more of the standards. Use of standards also extends the system lifetime by allowing convenient component upgrade as the technology changes. The standards that it uses include:

- o DoD Internet (IP) and Transmission Control (TCP) Protocols.
- o ARPA standard gateway.
- o Ethernet.
- o IEEE 796 bus (MultiBus).
- o UNIX Constituent Operating System

3.3 Software Architecture

The basic system organizing principal underlying Cronus is an abstract object model. With this model all system activity can be thought of as operations on a collection of objects managed by the system and organized into classes called types. Examples of

object types are files, processes, and directories of catalog entries. The type of an object defines the operations that can be invoked on it. The underlying structure of Cronus, which is largely hidden from client processes, consists of the primitives and mechanisms for delivering the operations to objects and delivering the results, if any, of operations back to the invoking client.

Cronus is implemented by a number of processes that reside on hosts which are part of the cluster. Some processes, called object managers, play a special role in implementing objects. Generally, when an operation is invoked on an object, it is delivered to a manager for the object which performs the operation. Other processes run application programs. Still other processes provide services and functions for users. For example, the user interface runs as a process

There are three interrelated parts to Cronus:

1. The Cronus kernel, which supports the basic elements of the object model: processes, communication between objects, object addressing, and the relationship between objects and their manager processes. The Cronus IPC is a major part of the kernel.
2. A group of basic object types, along with the object managers which implement them. The basic object types include files, processes, devices, and user records.
3. User interface and utility programs which provide convenient access to Cronus objects and services. The user interface and utility programs make use of the Cronus IPC and the basic objects to provide their services.

In addition, Cronus includes a set of rules for building and accessing new types of objects, which spell out the methods for integrating new object managers. Typically object managers for new types make use of the Cronus IPC and existing object managers to implement their new types. Application programs developed for Cronus may make use of existing types by means of the Cronus IPC, or may include managers for new types.

### 3.4 The Cronus Object Model

#### 3.4.1 Cronus Objects and Operations

The object model in Cronus provides a framework for both the system itself and application subsystems. Cronus system components, such as processes and files, are implemented as objects, and operations on them support system services. Application programmers are encouraged to use the object model for the standard access paths it provides to pre-existing objects, and for the facilities that are available to create new objects and object managers.

The definition of an object in Cronus is tailored to the distributed nature of the system. Special emphasis is placed on allowing efficient access to objects without detailed information about their current physical location.

All Cronus objects have several components:

1. A Unique Identifier ( UID ). A UID is a fixed length structured bit string guaranteed to be unique over the lifetime of the system. It serves as a handle for a particular object. It consists of a unique number or UNO and an Object Type field. The UNO guarantees uniqueness and incorporates the host upon which the object was created. The Type serves to classify the object.

Although ultimately, all references to objects are through UIDs, Cronus implements a symbolic name space which provides a mapping between user-defined symbolic names and object UIDs in order to facilitate user references to objects.

2. A Set of Operations. Processes may perform operations on an object by sending request messages to the object's manager. An object manager is a process or set of processes responsible for maintaining and manipulating an object. By convention, all managers are responsible for performing several generic operations on their objects. In addition they may perform any number of object specific operations.
3. An Object Descriptor. This is data associated with the object. It is maintained by the object's manager. It consists of several required fields and any number of

object specific fields. Some of the generic operations are defined for accessing object descriptors. Cronus achieves a consistent system model partly from the uniform integration and handling of these object attributes.

A mechanism is provided to group the objects associated with a particular manager. Each object has an associated Object Type. A manager may declare itself a manager of one or more object types. A functional area is typically supported by a set of functionally equivalent and cooperating manager processes distributed on various hosts of the system.

A useful property of type managers is that they may be accessed by simply knowing the object type that they are responsible for. A special UID is provided this, the generic UID of a given type. Generic UIDs are used for creating new objects and for status probes.

An object's type is also used to describe its attributes. In particular the set of operations associated with an object and the parameters necessary for each operation may be determined from the object's type.

The actual set of operations available on an object of a given type consists of the set of generic operations and a set of type specific operations. The generic operations are Create, Remove, Locate, and several operations which read and write standard fields in the object descriptors. Examples of type specific operations are Read (for files) and Lookup (for directories).

A key element of the object model is the Cronus kernel which supports communication between client and object manager processes. The kernel is message oriented, and it supports object-oriented addressing. When an operation is invoked on an object, the kernel delivers the operation (in a message) to the appropriate object manager. Messages corresponding to operations are sent as messages addressed to the objects. The object addressed is the operand, and the message data contains the operation and any additional parameters necessary to specify the operation. When the manager for the object receives the message, it performs the operation requested. Responses are sent as

messages from object managers to requesting clients.

When invoking an operation a process need not specify the host where the addressed object resides. To deliver the message, the kernel must determine the appropriate host using the object UID. In general, three somewhat different classes of objects are accessed through the kernel. These are:

1. Primal Objects

These are forever bound to the host that created them.

2. Migratory Objects

These are objects that may move from host to host as situations and configurations change.

3. Structured and Replicated Objects

These are objects which have more internal structure than a single "atomic" object. An example is a reliable (replicated) file which has a number of identical primal files as its constituent parts.

The important thing about primal objects is that, given the UID of a primal object, the kernel always knows where to find it, since the host it resides upon is coded in the UID. Non-primal objects may move from host to host or may be replicated at several hosts. The kernel uses an object location procedure to find non-primal objects. This procedure locates an object by means of a mechanism that broadcasts the generic operation Locate (as a message addressed to the object). This ensures that every manager for the object's type receives the Locate operation message. Because Locate is generic, it is defined for all object types and implemented by all object managers. Any manager that manages the object will reply, thereby locating the object.

### 3.4.2 Layered Structure

Cronus provides a set of facilities for the composition of messages and their transmission to provide a systematic communication facility among Cronus processes. There are three parts to this communication support:

- o An interprocess communication (IPC) transport facility, based on the object model and object-oriented addressing, provides Cronus primitives for uniform, host-independent communication among processes.
- o Conventions for passing data using Cronus canonical data types permit messages to be composed without concern for the heterogeneity within a cluster.
- o Protocols and conventions for constructing messages used in intercomponent interactions, especially the invocation of operations and the replies

The Message Structure Library (MSL) organizes these conventions and protocols by providing routines for the composition and examination of messages.

The IPC mechanism of Cronus is built upon the primitive functions Invoke, SendToProcess, and Receive. These primitives support the asynchronous communication of uninterpreted data octets among Cronus processes, by means of the abstractions of invoking an operation on an object or sending a message to a process.

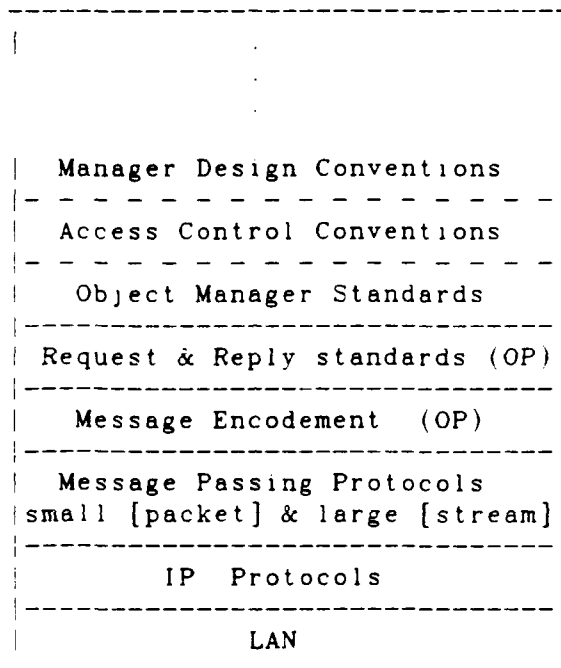
Messages, the entities communicated by the IPC, may be sent either reliably or with minimal effort. In addition, notions of both a small message which can be carried by a single datagram on the underlying transport mechanism, and a large message which may require an arbitrarily large number of datagrams are supported, although this distinction is hidden by the IPC library routines. Messages may be sent and received all at once or in pieces. The size of the chunk of data manipulated is independently selected by the sender and receiver. Large messages of indefinite size form the basis for interprocess stream communication

The Message Structure Library (MSL) is used to format messages, but is independent of the IPC. It provides a mechanism for inserting and extracting typed, structured data into a message buffer in a position- and machine-independent manner. Associated with the MSL are conventions, called the Operation Protocol, for the patterns of communication that arise in performing operations on Cronus objects.



Cronus is based on a layered architecture. This provides the opportunity to use only a subset of Cronus for a specific or limited application, and to easily replace individual parts of the implementation with alternative but equivalent implementations, should the need arise for optimization purposes.

A top down layout of the layers of the Cronus model follows.



The lowest levels consist of communication and message routing protocols. The next two consist of the definition and standards for using the Operation Protocol (OP) to send and encode requests and replies. Above these are the standards for implementing an object manager (including the set of required object operations). The highest levels consist of conventions for using the mechanisms provided in Cronus.

Although the Cronus system concepts are language independent, and implementations exist for other languages, we have concentrated our major implementation effort on the C programming language. A C language subroutine interface is provided to perform common tasks such as message generation and parsing. In addition to these, higher level tools are available for the development of new managers, including parameterized

routines for designing manager control structures.

The following sections describe each of the layers of the Cronus Model beginning with the lowest level protocols and continuing through to a brief description of the C language interfaces.

### 3.4.3 Message Passing Core

Process-to-process messages form the basis of all Cronus Operations. A Cronus Operation in the simplest case consists of a request message from a client to an object, and a reply message from the manager of the object sent back to the client. A complex operation may involve many sub-requests to various managers and replies to each of the requests.

The paradigm for a Cronus Operation is as follows: A client process determines the object UID on which it should perform an operation. Using the object and the operation it constructs and sends a request message:

<object uid, operation, data>.

The appropriate object manager receives the request in this form.

<client uid, object uid, operation, data>.

The manager services it and sends a reply message back which the client receives as:

<manager uid, reply data>.

It is important to note that the actual locations of the client, the object, and the object manager can be transparent to both. In order to support this flexibility in a heterogeneous environment, the lowest level of Cronus consists of host independent protocols for message definition and routing.

The Operation Protocol is a protocol designed for the host independent encodement of messages. Message data is encoded as a list of values of the form.

<data type, encoded value>.

A key is associated with each value to reference it. Encoding and decoding algorithms for each data type depend on the local host's internal data structures. OP defines many standard data types for values, such as integers and character strings. In addition, higher level constructs such as arrays and substructures are possible.

The Cronus Interprocess Communication (IPC) facility is used to deliver messages to the appropriate destination. An operation switch Peer To Peer Message Protocol is used to route messages between hosts. This protocol provides reliable routing, sequencing, and delivery of inter-host messages. In the case of communication failures, it also provides provisions for message rerouting and error notification.

#### 3.4.4 Primitive Operations

The primitive operations available to Cronus Processes are Send, Invoke, and Receive. Send is used to send a message directly to a process. Invoke is used to perform an operation on an object. An Invoke is delivered to the manager of a given object, based on its type. Receive is used to receive the next message.

A simple client/manager operation illustrates how these functions are used. First, the manager process waits in a Receive state for a message to arrive. A client process sends a request to the manager using the Invoke operation on one of the manager's objects. The client then typically does a Receive and waits for a reply. The manager services the request and Sends a response back to the client. The client gets his reply and the manager performs a Receive to wait for the next operation.

The distinction between the Invoke and Send primitives here is an important one. The client need not know the specific identity of a manager to direct a request to it using Invoke. The target of an Invoke is the object UID. It is the IPC mechanism that routes the message to the object's manager.

The separation of the Invoke operation from the Receive that generally follows it allows for complex asynchronous operations and optimizations involving parallel execution. In particular it is possible for managers to Invoke sub-operations while simultaneously being available to start new operations.

To encourage use of asynchronous message handling, a simple multitasking facility has been implemented for use within Cronus managers. New requests are associated with tasks, each of which may perform simple Invoke/Receive operations without complex state saving and restoring techniques.

#### 3.4.5 Object Location and Message Routing

A consideration that is unique to the distributed environment is the location of resources. It is often impossible to guarantee the availability of certain hosts in a configuration, yet it is desirable to use them when they are available. Cronus provides support for these specialized problems by defining objects which may be moved from one host to another, or which may be replicated on several different hosts, and by supporting a dynamic binding procedure for these objects.

Cronus objects types fall into three categories. Primal, Migratable, and Replicated. Primal objects remain where they are created, migratable objects may move from time to time, and replicated objects may be located on several hosts simultaneously. In this context, an object's location refers to the host on which a manager process may service requests directed to the object.

If a primal object's host is active and its manager process is running, it may be accessed. Primal objects are the simplest kind of objects, and they require no cooperation between the various type managers on the system which handle them. The IPC mechanism routes an invoke to a primal object by using the host of origin from the object's UID, and delivering the message to the appropriate manager process on that host.

The notion of a migratable object is somewhat more generalized. Migratable objects may move from one host to another. The object managers are responsible for much of the mechanism necessary to support migrating objects. They implement, by convention, manager to manager protocols for moving objects, and forwarding for misdirected messages to previously migrated objects.

The operation switch binds invocations to migratable objects by first broadcasting a Locate request to all potential managers of an object. The correct manager, if it is available, answers the request, and the message gets delivered to it. As an optimization, the current locations of recently accessed objects is cached.

Replicated objects are the most complex. A replicated object is maintained simultaneously by a number of manager processes. Its manager processes keep copies of the object data, which they synchronize by means of manager to manager communication. Invocation binding is handled in the same way as for migratable objects, the difference being that any of the available managers of a replicated object may answer locate requests. Currently the first one to respond is chosen for the operation invocation.

#### 3.4.6 Properties of Messages

The message passing primitives have a number of properties beyond transmitting datagrams. These consist of routing options, header information passed to the receiver, and support for error recovery. Invoke and Send functions include the following options.

- a. Maximum time. The maximum amount of time allowed for a message to be routed.
- b. Low Effort Option. Normally, messages are passed between hosts using a reliable transmission protocol. Messages which are passed for informational purposes often don't need guaranteed delivery, therefore a routing option is available which attempts to minimize the overhead in sending the message.

- c. Broadcast Option. When this option is selected, a copy of the message is sent to each host in the configuration. This important feature is used for object location and status queries.
- d. NACK Option. If this is set, a negative acknowledgement will be generated and delivered to the sending process if there is a timeout or routing error.
- e. Start Large Message. Requests that a direct connection be established between the source and destination processes to facilitate the transmission of large amounts of data.

When a message is actually delivered, the destination process receives, in addition to the message, a header consisting of:

- a. The sender's routing information. This may include the handshake data used to establish a large message connection.
- b. The actual target UID. Obviously this is mandatory for Invokes, since a single manager may manage many objects.
- c. The source UID. This is the basis for access control in Cronus. Before servicing a request, a manager may determine the requesting user and access rights associated with the requesting process based on its UID.

A C library interface exists for these primitives. It provides a uniform interface for small and large messages, as well as default values for all the routing options.

#### 3.4.7 Message Encodement

The Operation Protocol is used to encode Cronus messages. It is designed to facilitate the transfer of structured binary data between clients and managers. Message data is converted to a list of (key, type, value) triples of the form:

< <key1, type1, value1>, <key2, type2, value2>, ... >

where each value has been encoded into a machine independent

external representation, for the type and each key is used as a handle to reference a particular value. All messages are transmitted in this format. On the receiving end the message is decoded into a local format for the type which is compatible with the local architecture and programming language.

Message data is order independent; the only requirement is that each key be unique. The host independent design allows messages to be modified and resent without any data reformatting.

A set of C language library routines called the Message Structure Library (MSL) is provided to simplify the task of creating, modifying, and decoding OP messages. In particular these include: a PutMSValue function which encodes a value into its typed external format, associates a given key with the value, and adds the key value pair to a message being composed; and a GetMSValue function which finds the value associated with a given key in an OP message and converts it to internal format.

In addition to the application dependent values, OP messages by convention contain a number of standard key-value pairs. These are used by system tools to trace messages and analyze message traffic, and by high level standardized functions which are used to sort and match requests and replies. Standard keys include:

Global Request ID: A UNO which identifies an operation between a client and a manager; all messages associated with this operation including any necessary manager to manager message interactions are labelled with this identifier.

Request ID: Another UNO which identifies a specific request/reply interaction.

Message Type: Labels the message as a request, a reply, or simply an informational posting.

Operation (requests only): The object operation associated with a request.

General Reply Code (replies only): A general status indicator describing the degree of success of the requested operation.

The message passing, data encoding, and object binding mechanisms provide the Cronus system developers and Cronus application developers a powerful base for supporting remote invocations and remote procedure call type interactions in distributed computations.

#### 3.4.8 Extending Beyond a Local Network

Cronus makes extensive use of broadcast facilities provided by its communication base (typically a local area network or LAN) to locate object managers and objects dynamically. It is often true that a selected collection of hosts to be integrated into a Cronus system are not limited to a single LAN. In order to extend Cronus utilities across several local area nets and allow hosts which are not on a LAN to fully participate in Cronus, it is necessary to provide some means of forwarding broadcast packets between LANs or to the off-LAN hosts. Since Cronus IPC already uses internet addressing, regular (i.e. non-broadcast) messages can already traverse multiple networks without special handling.

Cronus uses broadcasts primarily to locate objects and for collecting status reports. Cronus broadcast packets are generated (and received) exclusively by the Cronus kernel (operation switch). One possible approach, therefore, is to put the broadcast-packet forwarding function into the operation switches. This approach limits the availability of multi-LAN broadcasts to Cronus, and further burdens and complicates an already busy and complex program. Another problem with using the operation switch to forward broadcast packets lies in configuration control: all the operation switches would have to know about all the off-net sites, or a subset of operation switches must take care of forwarding packets to off-net sites which were generated by other operation switches.

A second approach is to teach gateways about forwarding broadcast packets. Gateways already give one the correct access to the network, and there are proposals to do just this for the address resolution protocol (RFCs 917 and 925). The problem with this approach is that current gateways are not particularly flexible, nor do we have sufficient control over the gateway to be able to add new destination networks to its broadcast-forwarding tables. Also, in a future filled with many Cronuses governed by different agencies which may not own the gateways



involved (which in turn may be manufactured or programmed by several different companies) administrative control over this mechanism may present a problem.

A third approach is to build a separate program (which we call a broadcast repeater) which listens for broadcast messages on its LAN, and forwards the packets to other LANs. Such a program can be simple, flexible, and has application as a general network utility outside its use in Cronus. For example, many applications rely on broadcasting datagrams to distribute information. Rather than teaching these programs about a network composed of several LAN sub-networks, a network of broadcast repeaters can convey the information transparently across network boundaries.

The broadcast repeater system has two halves. a passive, listening half, and an active, broadcasting half. The passive half, on LAN A, listens for broadcast messages. When it receives a broadcast message (recognized by being a message addressed to the broadcast address of LAN A) it can filter it according to a variety of fields in the packet, and forward the packet to any active repeaters on other LANs which may be interested in this packet. An active repeater on LAN B then replaces the destination address with the broadcast address of LAN B (preserving the rest of the packet and recalculating the necessary checksums).

We have taken the broadcast repeater approach because of its flexibility and simplicity.

The broadcast repeater system serves as a transparent medium for relaying broadcast packets from one LAN to another, and also for forwarding broadcast packets to off-LAN hosts. To achieve this transparency the repeater requires access to the raw network layer, which may not be provided by many network implementations. Only one system per network need have this capability, however. An active repeater needs to be able to write an IP packet to the network with the same source address as the original datagram on the source network. The repeater must be able to specify the host of origin of the message in order to serve as a transparent medium.

A passive repeater may also forward the packet directly to a destination host (presumably an isolated host located off the LAN) rather than to a LAN repeater. In this case the passive repeater simply replaces the destination address with the address of the destination host and forwards the packet. Of course, since the packet is a UDP datagram, and since it is traversing gateways, it is not guaranteed that the packet will arrive at an off-LAN site. It is the role of higher levels of protocol to take the appropriate steps to insure delivery if that is desired. This is precisely the way UDP is specified, although experience on LANs has shown that very few datagrams actually do get lost.

Passive repeaters send packets to active repeaters through TCP connections. TCP provides guaranteed, sequenced delivery, and automatic notification of either end of the connection going down. The drawback of using TCP is that a single repeater can talk to a limited number of other halves: hosts limit the number of TCP connections a single process can hold. Fortunately, this limit is likely to be large enough for our application, and should it become a problem, then the use of TCP can be dropped in favor of a sequenced datagram protocol.

Another potential problem with this scheme is "ringing": passive repeaters forwarding re-broadcast packets back to their net of origin, where they are again forwarded to other nets. Ringing can be avoided simply by having the passive repeaters not forward broadcasts which have an origin off of their LAN. If a broadcast has an off-LAN origin it may be safely presumed that the packet got to this LAN via a repeater which forwarded it to all the places the packet should go.

### 3.5 Basic Cronus Types

The basic object types supported as part of the initial Cronus development effort and their corresponding object managers include:

1. Process objects and process managers that support the Cronus system and application processes.
2. User identity objects, called principals, objects which are collections of principals, called groups, used to support

user authentication and access control. These objects are managed by an authentication manager.

3. File objects and file managers that provide a distributed filing system.
4. Catalog and directory objects and catalog managers that implement the Cronus symbolic name space.
5. Device objects and device managers that support the integration of I/O devices into Cronus.

### 3.6 Access Control in Cronus

All client access to Cronus objects is subject to access control. The goals of the access control mechanisms are:

1. Prevention of unauthorized use of Cronus and unauthorized access to data and services maintained and provided by Cronus.
2. Preservation of the integrity of the system and its components.
3. Support for a uniform user view of access control to Cronus resources and functions.
4. Survivable authentication and access control in the presence of a wide range of host and communication failures.

The basis of access control in Cronus is the ability of the Cronus IPC to reliably deliver the identity of the invoker of an operation to the receiver of the message. The recipient of a request can then decide on the basis of the sending client's identity whether or not to perform the operation requested on the particular Cronus object.

For this to be a useful basis for access control there must be a means for reliably associating authorizations with clients. Mechanisms are required to establish bindings between client processes and authorities, and for object managers to determine the authority bindings for client processes.

Ultimately, most activity within Cronus is the result of requests initiated by users. Human users are represented internally to Cronus by objects of the basic type "principal". The authority bindings are, therefore, a correspondence between client processes and principals. By extending the notion of a principal beyond human users to include system elements, such as object managers, all activity in the system can be thought of as initiated by principals.

From the viewpoint of access control, the identity and authority of the principal (user) corresponding to a client process requesting an operation must be checked prior to performing the operation. Access control in Cronus involves two things:

1. Identification Authentication, determining the identity of the principal requesting a particular operation.
2. Authorization Verification, determining whether a given principal has been authorized to perform an operation on a particular object.

For example, when an object manager must decide whether to perform an operation, it must know the identity of the principal requesting the operation (Identification Authentication) and the rights the principal may have with respect to the operation and object (Authorization Verification).

Cronus uses access control lists to support authorization verification. In its simplest conceptualization, an access control list (ACL) is a list of principals that serves to limit access to an object for a particular action to those principals on the list. This simple idea is extended in two ways.

1. The UID for a group of principals may appear on an ACL. This makes it possible to authorize a group of principals rather than authorizing each individually. (Like principal, group is a basic Cronus type.)
2. A set of rights is associated with each UID on an ACL. There is a right associated with each possible operation. Each right in the set represents authorization to perform one or more particular operations. This makes it possible for an ACL to selectively control access to an object on a per-operation basis, and for the rights to be customized

for each type.

When a user (U) attempts to start a Cronus session, a process (P1) is allocated. The authority-binding for P1 cannot be established until the user demonstrates by the Login operation that he is U. The Login operation involves an authentication dialogue between the user and Cronus through which the user supplies a name and password that is checked against information stored with the principal object in the user registry. If the name and password are valid, the set of groups to which the user belongs is computed from a list of group UIDs maintained with U's principal record. Since groups can contain groups, this is a transitive closure computation. The user's principal UID is combined with the result of the computation to form a set called the access group set (AGS). The AGS is then bound to the authenticating process through its process manager. Processes subsequently created by an authenticated process inherit the AGS of the creating process.

In order to perform an access control check for an operation on an object, the manager for the object needs to determine the AGS binding of the client process. The identity of the client process is known to the manager because its UID is delivered by the Cronus Operation Switch along with the message that requests the operation. The AGS binding of the client can be obtained by invoking the BindingOf operation on the client process. After obtaining the AGS of the client process, the manager can perform the access control check by comparing the AGS with entries on the ACL.

When new objects are created, they are given an access control list which may be initialized under client control. There are generic operations which apply uniformly to all objects for further manipulating access control lists.

### 3.7 File System

Cronus supports a variety of different file types. Object managers for three types of files are part of the basic Cronus system design.

- o Primal files.

A primal file is stored entirely within a single host and it is bound to the host that stores it. Primal files provide mechanisms for supporting atomic updates and rollback to a prior consistent state. Other Cronus managers are a major user of primal files.

- o Fast files

Fast files are primal files without provision for atomic update, rollback or other internal consistency measures. As a consequence their performance is significantly better than regular primal files. Fast files have been used extensively when integrating existing software tools to the distributed file system environment.

- o Reliable files

Reliable files are implemented by one or more primal files. Each primal file used to implement a reliable file contains all of the file data. The reliability of these files derive from the fact that the file is accessible as long as at least one of the primal files that implement it is.

A reliable file can be moved from host to host. When a reliable file is moved, the primal file or files which implement it change.

Other types of files can be supported by developing appropriate object managers and using an appropriate data storage medium. For example, a type of "dispersed" file which had its contents distributed over several hosts could be implemented by several primal files each of which would be used to contain part of the contents of the file. Alternatively, Cronus can be used as a uniform and convenient remote access path to existing constituent operating system file systems and files. This is another aspect of the evolutionary nature of the Cronus system architecture.

### 3.7.1 Primal Files

The primal file system is partitioned among hosts that store primal files. Like other Cronus objects, primal files are accessible to processes by means of the Cronus kernel. There is a Primal File Manager process on each host that supports primal files.

A primal file is bound to the host that stores it. The host field of the UID for a primal file always specifies the host that stores the file. Hence the UID for a primal file always contains a valid "hint" identifying the host that stores it.

Primal files cannot be moved from one host to another. A copy of a primal file stored on one host can, of course, be created on another host, and the original can be deleted. However, the copy is a different primal file with a different UID which happens to have the same data as the original.

Most of the operations provided by conventional operating systems (create, read, write, etc.) are supported for primal files.

Cronus provides conventional access to files bracketed by open and close operations, and it also supports a type of access to files, called "free" access, for which bracketing open and close operations are unnecessary. "Open" and "close" operations are supported for situations that require reader-writer synchronization and to permit optimization of file i/o in situations where repeated read or write operations are performed. Supporting open/close requires that the File Manager maintain state information for open files and be prepared to deal with the problem of files that are opened and are never explicitly closed (e.g., because the client's host has crashed). Furthermore open and close represent significant overhead when only small amounts of data are to be written or read. Supporting two modes of access (open/close access and free access) enables a client to choose the mode appropriate for the situation.

Free reads and writes are synchronized in the sense that multiple reads and writes are serializable. This means that the File Manager in effect, performs each read or write operation in its entirety before performing another operation. A client

process may read or write data in a primal file (subject to access control considerations) without opening it, unless another process has opened the file in such a way that free reads and writes are forbidden.

When a file is opened, two parameters specify the access state requested. One specifies either Read or ReadWrite access. The second specifies the type of reader-writer synchronization desired. There are two types of synchronization supported: "frozen" which permits either N readers or a single writer; and "thawed" which permits any number of simultaneous writers and readers. When a file is opened with "thawed" access, readers of the file see updates made by writers of the file. Opening a file with "thawed" access prevents other processes from opening it "frozen".

A file may be opened so long as the access state requested does not conflict with the current access state of the file. The access states defined for a file are:

- idle;
- frozen read open;
- frozen readwrite open;
- thawed open;
- (free) read in progress;
- (free) write in progress.

Table 1 defines the compatibility of the access states with client open, read and write operations. An OK for an (OPERATION, ACCESS STATE) entry in the table means that a client process can perform the operation on a file when the file is in the corresponding access state; a NO entry means that the operation will fail when the file is in the corresponding state; a DELAY operation means that the operation will be delayed until the operation in progress (and any others that may be queued) are completed.

In order to support file system recovery, data that is written to a file that has been opened for (ReadWrite, Frozen) access does not become part of the permanent file data until the file is closed. It is possible to close a file opened for



ACCESS STATE

		idle	frozen	frozen	thawed	read	in
			read	readwrite		progress	
	OPERATION						
write in progress	frozen read	OK	OK	NO	NO		OK
DELAY	open						
	frozen readwrite	OK	NO	NO	NO		DELAY
DELAY	open						
	thawed	OK	NO	NO	OK		DELAY
DELAY	open						
	free read	OK	OK	NO	OK		OK
DELAY							
	free write	OK	NO	NO	OK		DELAY
DELAY							

Access State Compatibility  
Table 1

(ReadWrite, Frozen) access in a way that aborts writes made to the file while it was open.

When a process is destroyed with files open, the files are closed and any writes to (ReadWrite, Frozen) open files are aborted. The normal close operation may only be invoked by the process that opened the file. An alternate close operation can be used by other processes to close a file during cleanup.

The Primal File Manager does not acknowledge write requests until the data has been written to non-volatile storage. A client process can be sure that the data has been written when the acknowledgement is received, even if the Primal File Manager or its host should crash shortly afterward.

Primal File write operations are atomic with respect to host crashes. That is, if the Primal File Manager host should crash during a write operation, after the host and Primal File Manager have been restarted and the Primal File Manager has performed its recovery procedures, the write operation will have either occurred in its entirety or no part of it will have occurred. If the crash occurs after the data has been safely written but before the acknowledgement has been sent, the acknowledgement will never be generated.

As with other object types, access to a primal file is controlled by its access control list (ACL). Access to a primal file may be granted to other users by adding entries to the ACL. Similarly, access to a file may be revoked from a user by removing the corresponding entry from the ACL.

### 3.7.2 Reliable Files

The principal motivation within Cronus for maintaining multiple copies of a Cronus object derives from reliability considerations. The objective is to increase the probability that the object will be available for access at any given time by keeping copies (in Cronus we shall call them images) of the object at a number of hosts. Although any given host that stores the object may fail, so long as at least one of the hosts maintaining an image is accessible, the object will be.

Secondary benefits include performance improvements that may result from distributing the object access load among the hosts that store it, and from the possibility that client access to an image maintained on its own host will be more responsive than access to an image on a remote host.

Increased object availability does not come for free. The cost is increased complexity in managing the object. Most of the complexity is a consequence of the fact that the system works to ensure the mutual consistency of the images; when one image changes, all others should be updated to reflect the change.

Reliable files are an example of a replicated Cronus object. In Cronus, different approaches to handling the coordination and consistency of replicated objects are possible for each different object type. These details for different forms of replication are handled within the managers for the object type in question. In this section we discuss the design considerations for reliable file objects.

Concurrency control requires that sites managing images of a file cooperate to synchronize client access to the file. In addition, since strong concurrency control mechanisms require the participation of more than one site, situations may arise where an insufficient number of file image sites are accessible to perform the concurrency control. Unless the system is willing to permit unsynchronized access to an accessible file image in such situations, some of the reliability benefits of multi-image files will be lost. The danger of unsynchronized access is, of course, that accessors may cause different images of a file to become inconsistent.

The approach to concurrency control for reliable files is based on the presumption that file availability is important enough that it is permissible to risk the consistency of file images and to grant access to file data when synchronization cannot be achieved. That is, when a choice must be made, file availability or survivability is considered more important than mutual consistency of file images. The reliable file manager will try to achieve strong synchronization prior to file access in order to maintain the consistency of the file images. However, should the synchronization fail because the file sites required to achieve it are inaccessible, the client will be informed and access to the file will be permitted only if the client gives

explicit consent to continue.

This approach to concurrency control will be practical only if:

1. File access patterns are such that it is relatively unusual for multiple concurrent updates to occur.
2. Hosts are reasonably reliable so that host failures that prevent strong synchronization are relatively rare.
3. There is a simple and inexpensive way to detect inconsistent images of a file. We believe that the Version Vector mechanism developed at UCLA [Parker1983] is a good one for this purpose.

A Cronus Reliable File (RF) is a collection of one or more primal files, each of which represents an image of the reliable file. No two images of a reliable file are stored at the same site.

The number of images of a reliable file may change over the lifetime of the file, as may the sites which maintain the individual images. The desired number of images is called the cardinality of the file. The actual number of file images may be different than the file cardinality. For example, when a file is first created its cardinality will be greater than the number of images until all of the images are created. Similarly, if the cardinality of a file is changed, it takes finite amount time for the number of images to be adjusted. Thus, the cardinality is properly thought of as an objective.

Each Reliable File Managers (RFMs) maintains a UID table for the reliable files that it manages. Unlike simpler objects, such as primal files, the management of reliable files requires the cooperation of RFMs. Each RFM participates in the management of a collection of reliable files (the ones in its UID table), but not all RFMs participate in the management of all reliable files.

When a client invokes an operation on a file, the underlying interprocess communication facility routes the operation to an RFM capable of performing it. Any interactions among RFMs that are required to perform the operation are transparent to the

client process.

Access to the primal files that comprise a reliable files is limited to RFMs. No other process may directly access a primal file used to implement a reliable file, even if the process has the UID for the primal file; this is enforced by the Cronus access control mechanism. RFMs reside only on sites that also have primal files managers (PFMs). The manager's image of the file is stored at the manager's site.

In order to maintain the consistency of images of reliable files and the integrity of internal file data (for primal as well as reliable files), the manner in which clients access the files must be controlled and synchronized.

The reliable file approach to synchronization can be characterized as a best effort approach consisting of the following steps:

1. try to synchronize access;
2. if synchronization cannot be achieved permit access if the client so desires;
3. be prepared to detect and deal with inconsistencies that may result from unsynchronized access later.

A specific concurrency control mechanism must be chosen. Although much has been written about concurrency control and synchronization for multiple copy files and data bases, there is little practical experience on which to base a choice. We have decided to use a simple mechanism at least initially. Should the mechanism prove to be inadequate (for example, because it cannot achieve synchronization often enough, given the failure patterns observed in Cronus), it can be replaced with a more capable (and complex) one.

Synchronization will be accomplished by means of a primary/secondary image approach. Each reliable file will have one primary image and one or more secondary images. All attempts to synchronize access to a reliable file will require synchronization with the primary image. We refer to the manager of the primary image as the primary manager for the file;

managers of other images are called secondary managers.

When a client attempts to access file data in a way that requires synchronization, an attempt will be made to synchronize with the primary image of the file. If the client's access attempt is initiated with the manager for the primary image, synchronization occurs as for primal files. If the access attempt is initiated with the manager for a secondary image of the file, the secondary manager interacts with the primary manager to gain the appropriate kind of access (non-exclusive read, exclusive write).

RFMs use a locking discipline to support synchronization. This discipline works roughly as follows. When an attempt to open a file for reading is handled by a secondary manager, the manager tries to set its lock for the file to "reserved for reading". The attempt to set the lock fails if the file is already locked for writing. Next, the manager interacts with the primary manager to try to set the primary manager's lock for the file. If this succeeds, the secondary manager sets its lock to "locked for reading" and proceeds with the open. If the primary has the file locked for writing, the secondary manager clears its lock and reports to the client that the file is busy. When the file is closed, both the local lock and the primary manager's lock for the file are cleared. Attempts to open a file for writing are handled in an analogous fashion.

If synchronization for any operation fails because the primary manager cannot be reached, the operation may proceed, but only with the explicit consent of the client, and, of course, at some risk. The risk is that different images of the file may be undergoing unsynchronized access, and, as a result, the file images may diverge into inconsistent states.

A client may specify its intent with regard to unsynchronized access when it initiates a file operation by means of an optional operation parameter. Alternatively, the client may choose not to specify the action to be taken when it invokes the operation, in which case, if synchronization cannot be achieved, the manager will ask whether it should proceed with or abort the operation.

Inconsistent images of a file can be detected by means of a version vector mechanism developed at UCLA. A version vector for a reliable file, RF, is a set of N ordered pairs, where N is the number of sites at which RF is stored. A particular pair  $(S_i, V_i)$  counts the number of times updates to RF were initiated at  $S_i$ . Thus, each time an update to RF originates at  $S_i$ ,  $V_i$  is incremented by one. The version vector is part of the object descriptor for RF.

Two images of a reliable file are said to be consistent if the modification history of one is the same as or is an initial subsequence of that of the other. It can be shown that two images are consistent if one of the vectors is at least as large as the other in every  $(S_i, V_i)$  pair. The larger vector is said to dominate the smaller, and the image corresponding to it represents a later, consistent version of the image corresponding to the smaller vector. If two vectors are such that neither dominates the other (that is, some pairs in one are larger than some pairs in the other and vice versa), then the corresponding file images are inconsistent with one another.

RFM's must interact with one another in order to maintain reliable files. For example, when a reliable file is updated, the new file data must be transmitted to each site that has an image of the file.

Occasionally a RFM that must participate in such an interaction will be inaccessible. It is important that when, if ever, such a RFM becomes accessible the interaction occur. It is the responsibility of the initiating RFM to ensure that the interaction occurs.

The operations supported for primal files are also supported for reliable files. Three additional operations are supported for reliable files. The Change\_Cardinality operation changes the cardinality of a reliable file. The File\_Sites operation produces a list of the sites that are thought to be maintaining images of the file, with the primary file site distinguished. The Move\_Image\_To\_Site operation moves a file image from one site to another (removing the image at the source site).

To create a reliable file, the client invokes the Create operation specifying the cardinality of the file as a parameter. The RFM that receives the Create operation becomes the primary manager for the file.

When a reliable file is first written and whenever the file cardinality is increased, the RFM selects sites to store images of the file. The acquisition of new sites involves three steps.

1. The selection of the new sites.
2. Obtaining commitments from the RFMs at the selected sites to store images of the file.
3. Updating file descriptors at each of the file sites to reflect the new sites.

The RFM at which write operations are performed is responsible for distributing updates to the other file images. It does this by interacting with the other RFMs sites in the following way:

1. It increments its (Site, Version) element of the file version vector.
2. It attempts to interact with each other RFM that manages an image of the file.
3. Should it fail to complete the image update with any RFM, it adds a record to a PendingActions data base for completion at a later time, specifying the file and the RFMs it was unable to update.

Version vectors are used to detect inconsistent images of reliable files. In the current design, both the descriptor for a file and the file itself are protected by version vectors.

Version vectors are compared in two situations:

1. When an image of a file is updated. The RFM initiating the image update supplies its version vectors, and the responding RFM compares them with its own.
2. When an attempt is made to lock a file for read or write



access. The secondary RFM attempting to lock the file supplies the primary RFM with its version vectors and the primary RFM does the comparison. See the Cronus System Subsystem Specification [BBN Report No. 5884] for more details on reliable files and the use of version vectors.

### 3.8 The Cronus Catalog

Cronus supports two system-wide name spaces for referencing objects.

At a relatively low level there is the name space of object UIDs supported by the Cronus kernel and object managers. Every Cronus object has a UID. Each object manager maintains a record of UIDs for objects it manages in a UID Table. When a manager creates an object it creates an entry for the new object in its UID Table. Each manager's UID Table defines a part of the UID name space. The entire Cronus UID name space is defined by the union of the UID tables of all the object managers. Thus, there is no single identifiable catalog of UIDs supporting the UID name space. Rather, the Cronus UID name space is implemented in a distributed fashion with each object manager responsible for implementing part of it.

At a higher level there is a symbolic name space for Cronus objects. The implementation of the symbolic name space is supported by the Cronus Catalog. The principal function of the Cronus Catalog is to provide a mapping between the symbolic names that people use to refer to objects and the UIDs that are required to actually access the objects.

Access to objects is supported by means of the invocation mechanism of the Cronus kernel. Typically access to an object will be initiated in one of two ways:

1. Directly through the UID name space.

The accessing client process has the UID of the desired object and invokes an operation upon it. The Cronus kernel delivers the requested operation along with the UID and any other parameters to the appropriate object manager. The object manager consults its fragment of the UID Table

to access the object as necessary to perform the requested operation.

2. Through the symbolic name space.

The accessing process has a symbolic name for the object. In this case, access is accomplished first by consulting the Cronus Catalog to find the UID for the object named. This involves a name lookup operation using the catalog. If successful, the lookup finds the catalog entry corresponding to the name which contains the UID for the object. With the UID for the object, access to the object can proceed as described in (1) above.

In either case, access control is performed by the object manager responsible for the object.

An object may have zero, one, or more symbolic names. When an object is given a symbolic name, an entry for the name is made in the Cronus Catalog, and when the name for an object is removed, its entry is removed from the Cronus Catalog. The Cronus Catalog is a Cronus object which is managed by the Cronus Catalog Manager.

Symbolic names are location independent in the sense that a name for an object is independent of its host location within Cronus and that a name that refers to an object may be used regardless of the location within Cronus from which it is used. In addition, symbolic names are uniform in that common syntactic conventions apply to names for different types of objects (including file, groups, etc.).

The symbolic namespace is structured hierarchically as a tree, much like the UNIX and Multics file name hierarchies. However, in Cronus any object may be given a symbolic name. The tree contains nodes and directed labeled arcs. Each node has exactly one arc pointing to it, and can be reached by traversing exactly one path of arcs from the root node. Nodes in the tree represent Cronus objects which have symbolic names, and non-terminal nodes correspond to directories which are objects implemented by the Catalog Manager.

The complete name of a node, and a symbolic name for the corresponding object, is the name formed by concatenating the labels on the arcs traversed on the path from the root node to the node in question. The syntax for a complete name is:

$$: \{ x : \}^* y$$

where "x" and "y" are arc labels, the "{","}" brackets indicate optional presence, the ":" is a punctuation mark to separate name components, and "\*" is the Kleene star.

It is also possible to name nodes relative to a directory. Such a relative or partial name is formed by concatenating the labels on the arcs traversed on the path from the directory in question to the node. The syntax for a partial name is:

$$\{ x : \}^* y$$

The Cronus catalog also supports "links". The catalog entry for a link identifies another point in the symbolic name space called the link target. The catalog entry holds a complete Cronus symbolic name for the link target. Links are cataloged as terminal nodes in the name hierarchy tree.

In addition to the generic operations, the Catalog Manager supports Enter, Lookup, and Remove\_entry operations. The Enter operation establishes a symbolic name for a Cronus object. Lookup interprets symbolic names. The Lookup operation is performed by using the Cronus Catalog in a straightforward manner. It begins with a designated directory (the root for a complete name or an implicitly or explicitly specified directory for a relative or partial name. Directories are used to evaluate the components of a name until either the last component of the name is consumed and its catalog entry is found in which case the lookup succeeds, or a name component cannot be found in a directory in which case the lookup fails. The catalog entry corresponding to a symbolic name includes the UID of the object named.

For some types of objects it is useful to be able to think of a collection of the objects as a sequence of "versions" or "revisions" of the same logical object. The Cronus Catalog supports versions for certain object types. For types for which versioning is supported, the Enter operation permits the same

Name to be entered into a given directory more than once. The first time a Name is entered the result is version 1 of the object. Subsequent entries of the same Name result in successively higher versions of the object. All of the catalog operations which take a name parameter allow the specification of a version number extension to the name, with appropriate defaulting in their absence.

### 3.8.1 Implementation of the Cronus Symbolic Catalog

The Cronus Catalog is implemented in a distributed fashion by a collection of Catalog Managers on several Cronus hosts. The following are some design considerations for the Cronus catalog

1. The catalog shouldn't be stored at ONLY ONE site.

Reliability consideration.

This implies that the information in the catalog should be distributed and possibly replicated in some fashion.

2. The entire catalog shouldn't be stored at ANY SINGLE site.

Scalability consideration.

This implies that the catalog should be dispersed among several sites in some fashion.

3. It should always be possible to access an object when the site that stores the object is accessible.

Reliability consideration.

This implies that the catalog entry for an object (or a copy of it) should be stored at the same site as the object. In addition, there should be sufficient information at that site to enable it to selectively control access to the object.

The UID Table exhibits this property. The Cronus symbolic catalog should also.

4. There is little utility in maintaining a catalog entry for an object in a more reliable fashion than the object itself.

Common sense consideration.

This suggests that there is little utility in replicating catalog entries for objects beyond that required by (3).

A directory is a collection of catalog entries. Directories are implemented by Cronus files. Directories are Cronus objects with symbolic names. The UID in the catalog entry for a directory is the UID of the directory.

Cronus files are stored in their entirety within a single host. Therefore, a directory is stored in its entirety within a single host. This means that the smallest unit of dispersal for the catalog is the directory.

The LookUp operation involves following branches corresponding to components of a Cronus symbolic name through a number of different directories. The location of the root directory, the start point for the lookup, is known to the Cronus Catalog software. With no further restrictions on the dispersal of the catalog the name lookup could require following branches (entry names) through a number of different directory sites.

It is desirable to place further restrictions on the dispersal of the catalog in order to limit the number of sites that must be involved in a lookup operation. A useful restriction is to

1. Require that the catalog structure for entire subtrees below a certain cut (the "dispersal cut") through the catalog tree be stored within a single site. We call a subtree that is rooted at the dispersal cut a "dispersal subtree".
2. Require that the catalog structure above the dispersal cut be stored within a single site. We call the structure above the dispersal cut the "root portion" of the hierarchy.

The first restriction ensures that lookup operations within a subtree that is below the dispersal cut can be confined to the site that stores the catalog portion corresponding to the subtree, and the second ensures that determining the site that

stores the catalog portion for any given dispersal subtree can be confined to the site that stores the root portion of the hierarchy.

The impact of these two restrictions is that lookup operations require at most two catalog sites.

We now observe that it is useful to replicate the root portion of the catalog hierarchy. Furthermore, a good way to replicate it is to maintain it at each site that maintains a dispersal subtree. The reasons for doing this are.

1. To distribute among several sites the load resulting from lookup operations.
2. To allow some lookup operations to be confined to a single site.
3. To increase the availability of the root portion of the hierarchy.

For this to be a practical dispersal, it must be possible to maintain the various copies of the root portion of the hierarchy in a mutually consistent fashion. A mechanism for maintaining this consistency is described in the Cronus System/Subsystem Specification. It is based on the observation that in many multi-user systems the root portion of the hierarchy changes only very slowly over time, and in quite limited ways. This is typically so because only a few users are authorized to make changes to the root portion, and because changes generally occur as the result of the addition or deletion of a user or project. This means that the mechanism need not be powerful enough to handle the most general form of the multiple copy update problem.

For considerations 3 and 4, the objective is to ensure that an object is accessible symbolically whenever the site that stores the object is.

The primary symbolic access path to a file is:

Symbolic name --> Cronus --> UID --> UID --> object  
Catalog Catalog Table

The problem to be addressed is how to handle the situation when the Cronus symbolic catalog is inaccessible.

There seem to be two approaches to this problem:

1. Replicate the catalog sufficiently to ensure that it is available with the degree of reliability that is desired.
2. Recognize that not every object will require the same degree of reliability, and replicate the catalog information required to access a particular object (i.e., its catalog entry) to the degree desired and store it at the site that stores the object.

In Cronus we are currently using approach (2). The idea is to maintain a secondary symbolic access path to objects. The secondary access path is supported at each object managing host by collections of copies of Cronus Catalog entries. The catalog system software is responsible for maintaining the consistency between the distributed catalog entry copies and the Cronus Catalog.

Under normal conditions, a symbolic reference to an object is accommodated by a Lookup using the Cronus Catalog in the normal fashion, following catalog entries from one directory to another until the Lookup either terminates on an entry for the symbolic name or fails because there is no entry (i.e., the name is not catalogued).

In situations when the catalog is unavailable, the secondary symbolic access path would be used for the Lookup. The Lookup would succeed whenever the object itself can be reached, since a copy of the catalog entry for the object is stored at the same site as the object, if the object has a symbolic name.

One can ask why not always use the secondary access path since it will always succeed when the object is accessible. The answer is that a Lookup by means of the primary path is "directed" whereas one by means of the secondary path is "undirected" (e.g., there is no a priori knowledge of which host or hosts should be consulted to perform the Lookup and it is likely that when the host is found the name to catalog entry mapping will take longer to perform since it may be difficult to

structure the search though the catalog entry copies.

### 3.9 Automating Cronus Manager Development

In previous sections we have discussed the elements of the Cronus object system support. As these parts of the system became available, the first Cronus object managers (file and catalog) were coded by making calls on the library routines previously mentioned. With this experience we easily recognized that much of the effort that went into developing a manager, and client software to access that manager's objects fell into a predictable pattern, was repetitious, and was largely the same from object type to object type.

With this in mind, we initiated an effort to elevate the abstractions which an application object manager developer uses. We provide for specification driven automatic code generation for much of the object framework which is common to all managers. This "manager compiler" automatically generates code to handle invocation, message receipt, parsing, dispatching, reply generation, access control etc. In sum, the application developer need only deal with and provide code for the problems (operations) which are specific to his application. The rest of the software needed to handle all of the intermediary details is automatically provided as part of the manager structure itself. In the rest of this section we discuss the current automated facilities provided to application developers for developing new Cronus object managers.

#### 3.9.1 Manager Facilities Provided Automatically

One begins development of a Cronus application manager by defining the various object types, the operations upon them, and the access control constraints. The criteria for determining exactly what should be objects are not by any means absolute, but often times objects will correspond to "real world" entities or abstractions pertinent to the user of the software.

In many cases, applications make use of existing data outside the Cronus environment. The data abstraction features of the object model favors this, since only the manager of a given



object is aware of its internal organization. Representations may even vary between managers of the same object type on different hosts (thereby providing a convenient mechanism for dealing with similar data from different sources).

Once defined, the object definitions are coded in a non-procedural specification language, compiled, and stored in the "protocol database". The database is then used to generate the code for the "automatic" components of the manager.

#### 3.9.1.1 Multiple Object Types

An implementor may choose to have a single process manage multiple, related Cronus object types (such as Principals and Groups, or several varieties of files). This can be advantageous with request to code sharing, concurrency control, faster access between data structures, or reduced process contention. The manager development software allows any number of types to be managed by a given manager. The mix is functionally transparent to the rest of the system, and may even vary between hosts.

#### 3.9.1.2 Dispatching

The actual tape dependent processing for an operation is performed by an implementor-supplied procedure. Dispatching to the appropriate procedure (based on the operation requested and the object type) is done by the manager software after the implementation-independent processing described below has been performed.

A given operation processing routine can be used in the implementation of more than one type (assuming, of course, that the operation parameters and semantics are equivalent). The processing routines are currently all written using the standard C programming language.

### 3.9.1.3 Multitasking

In order to facilitate interleaving of operations and to minimize the amount of time to initiate processing of an operation, the manager software supports a coroutine-style tasking facility, whereby multiple operations are processed simultaneously. The dispatcher (itself a task) creates a new task for each incoming operation.

To ease concurrency considerations, the tasking package is non-preemptive. A task will relinquish control to another task only by explicitly doing so, or when it awaits a reply from a (nested) operation invocation. The latter could occur, for example, during access to a Cronus file used in the internal object representation. It also occurs when obtaining the client's bindings during the access control check.

### 3.9.1.4 Access Control

Each Cronus object has an access control list associated with it, defining the access rights available to individual users or groups of users. The application developer may declare that the client must have a specified set of access rights in order to invoke a given operation on an object. Possession of this access will automatically be checked by the manager software. Of course, the application is free to impose additional procedural constraints and checks, and may reference the access control list directly.

### 3.9.1.5 Inheritance of Operations

The manager software supports "inheritance" within a hierarchy of types, with respect to both operation parameter definitions and code sharing.

This is the mechanism used to support the generic operations defined for all Cronus types. These are defined for type CT\_Object (the top of the type hierarchy), and include operations for locating the object or manager, obtaining descriptive information about it (available operations, interpretation of

access rights, etc), manipulation of access control lists and attributes, and status information. Although functionally distinct, these may effectively be viewed as part of the manager software.

#### 3.9.1.6 Message Parsing and Validation

The manager software attempts to shield the application from the unwieldy data representations required for network communication in a heterogeneous environment. Operation "arguments" are passed to the processing routine in internal programming language representations. Appropriate supplementary information is also provided for arrays, variable-length, and optional data elements.

The manager software also provides argument validation at what might be called the lexical and syntactic levels. This includes checking that required arguments are present, and that data types match. Of course, additional validation may also be performed by the application.

#### 3.9.1.7 Storage For Instances of Objects

Virtually every Cronus object has some type specific data associated with it. The manager software provides for efficient automatic management of such data, including retrieval each time the object is referenced. Both fixed and variable-length components may be specified. Of course, such data may also contain pointers into external data spaces known only by the implementation.

Such automated storage management will eventually include mechanisms for generic backup/recovery, replication, and migration of objects.

### 3.9.2 Client Facilities Provided Automatically

Support is also provided for developing the client software necessary to manipulate the objects. Of course, any manager may also be the client of other managers.

#### 3.9.2.1 Subroutine Interfaces

The manager software automatically generates subroutine stubs encapsulating the necessary argument processing (linearization and collecting data into the message body), operation invocation, response argument parsing, and error detection involved in an operation invocation. This effectively provides an RPC-like interface to all Cronus operations.

#### 3.9.2.2 Generic User Interfaces

The manager software also generates tables driving a generic, technology-retargetable user interface subsystem. This allows the direct command level invocation of any operation defined in Cronus. It is tremendously valuable as a debugging aid, and has also proved quite useful as a standard user interface (encapsulated slightly via command scripts) for those operations which map directly to user-level commands.

### 3.9.3 Documentation

Once annotations have been provided to the object definitions, the manager software can generate formatted descriptions suitable for inclusion in hardcopy system documentation (the entire section 3 of the Cronus Users' Manual was produced in this fashion).

### 3.9.4 Experience to Date

We have already used the automated manager development tools to generate a number of Cronus object managers which are in daily use, including the Cronus Authentication Manager. We estimate that using the automated tools we reduce the lines of code needed to be written by the application developer to about 1/6 that of a hand coded manager, without any noticeable change in performance. As Cronus itself is extended in areas of resource management and survivability, we anticipate including "off-the-shelf" and customizable approaches to these object attributes as part of the manager development package.

## 3.10 Cronus Monitoring and Control System

### 3.10.1 Role of the MCS

This section describes the existing and planned monitoring and control system for the Cronus distributed operating system. The Monitoring and Control System (MCS) includes monitoring and control of hosts, of the Cronus managers on those hosts, and of network communication. The monitoring and control station provides the functionality of an operator's console for the Cronus Distributed Operating System. The MCS treats Cronus as an integrated system, decomposed by function rather than by host. The MCS is designed to be integrated with and supports the abstract object orientation of the overall system architecture.

In their role as caretaker, operators use the MCS to review resource usage, to examine status and trouble reports from the services, to monitor host and peripheral device availability, and to activate and deactivate managers during routine hardware maintenance. As system specialists, operators use the MCS to relocate managers, to modify policy parameters that influence resource allocation decisions, to evaluate the effect of changes in policy parameters, to centrally monitor experiments, and to diagnose system problems.

Where practical, the MCS also monitors and controls Constituent Operating System (COS) functions, the processor and peripheral hardware and the network from the same station, but such functions are limited by our desire to modify the constituent software as little as possible.

Cronus is restarted from the Monitoring and Control System. For hosts, the MCS supports resetting the hosts and starting Cronus. For some hosts, such as those without disks, the MCS will download an executable image to start Cronus. For other hosts, the MCS will invoke programs executed by the host's constituent operating system to start Cronus.

### 3.10.2 Functional Areas

The following describes typical activities currently envisioned for the Cronus operator and MCS software.

#### 3.10.2.1 Fault Detection

Problem areas can be first identified by the operator, a component such as a host or manager, or by the MCS software itself. Operators recognize problems either from a report by another user, by having the MCS software detect anticipated fault conditions, by personally noticing unexpected system behavior, or by examining data collected by the MCS. The problem is then reported to the MCS and if the MCS cannot correct the problem itself, the operator is prompted to take action. An audible alert accompanies the report of a critical event.

The MCS and manager software detect problems in a number of ways. These strategies include a report from another system component or user, an unacknowledged polling or other request, or violation of a MCS recognized system parameter constraint. Priorities are assigned by the originator of system and MCS generated reports to guide the MCS and operator in scheduling their review.

The forwarding of fault messages from system services to the MCS forms the simplest MCS problem detection strategy. The warning is forwarded to the operator for review, sometimes with additional information supplied from the configuration database or from recorded data regarding the defective component's immediately past behavior.

The MCS periodically polls all system components to see if they respond and to determine their current status. Component failures are reported, along with the time the failure was first discovered. If the failure occurred recently, the component can be restarted from the MCS. Status polling and manager control

functions are carried out using standard Cronus generic operations, making it easy to extend monitoring and control functions to new resources and application "types" which are subsequently added to the system.

Thresholds and other constraints are also applied to collected system parameters, measured rates and resource usage. When a constraint is violated, a report to the operator is generated. For example, the operator is alerted if too many authentication failures occur, since that may indicate an intruder. More common situations, such as rising disk usage or high soft error rates, often foreshadow more serious upcoming problems, and can be detected and reported this way.

#### 3.10.2.2 Logging

Various data reported to and collected by the MCS is recorded. This allows long term statistical evaluation and comparison. It also provides a journal for reviewing events that led to a system failure. For example, health and status reports are recorded by the system. This information can be viewed later by an operator or included in reports. The contents can be analyzed for performance evaluation.

#### 3.10.2.3 Fault Isolation

A more difficult problem occurs when a program fails unexpectedly. The cause of the problem could be: a bug in the program; insufficient access rights by the client or a manager for some data or resource needed to satisfy the request; IPC, network or host congestion triggering timeouts; component failure leaving a non-reliable resource inaccessible; and so forth.

Using the MCS for fault detection, the operator starts with a high level view of the system, possibly narrowed by some initial sense of what caused the problem. For example, if the problem is frequent access request denials, the operator might start by examining the catalog and authentication managers as a whole functions to see if the problem was system-wide. Then, the particular managers servicing the request could be examined, to determine their current state. Then the processors and process managers running the managers might be checked, or perhaps the behavior of the Cronus kernel on the effected hosts depending upon what was causing the failure. The MCS software supports all

of these individual "views" of the system, and provides a convenient mechanism for interactively moving between different views as well as for constructing additional special purpose views.

#### 3.10.2.4 Fault Correction

A problem can be corrected at several levels. The MCS may be able to identify and correct the fault automatically. For example, if the MCS notices a failed manager, it can be restarted automatically. The restart operation is reported to the operator. If the restart rate exceeds some threshold, the MCS either automatically or under operator control could take the host offline from the MCS station.

When the MCS is not programmed to handle the failure, the operator must correct the problem. The MCS alerts the operator, provide information and guidance, and accepts and invokes commands from the operator to correct, eliminate or bypass the problem area.

Finally, there are the problems that require attention by the vendor. Hardware component failures and software bugs are both situations where the operator may not have the necessary resources to correct the problem and other specialists may be required. The MCS will, however, allow the operator to take the component out of service until the problem has been corrected.

#### 3.10.2.5 Resource Allocation and Policy Management

The operator can control managers and the resources they manage. Managers can be started or stopped, replicated to increase redundancy (provided they implement the appropriate consistency model) or relocated to a different host. The operator can adjust policy parameters that control the placement of new object instances such as files, the size of caches, quotas limiting the activity of a particular principal or manager, or can instruct the managers to relocate instances of migratable objects. One of the major roles of polling is to collect periodic snapshots of resource allocation and consumption patterns in order to coherently display these to the operator on either a collective functional or individual host basis.



The MCS maintains recent historical views of the monitored parameters to help the operator identify trends in the affected parameters. Based on these representations of immediately past resource allocation patterns, or based on values exceeding some predefined thresholds, the operator can adjust the resource management policy for the effected resource (see Cronus Resource Management).

### 3.10.3 Current Implementation

During the initial phase of Monitoring and Control System development, we implemented a set of functions to retrieve manager and host status information and several portable programs to display the data on common terminals. These programs run as clients on many Cronus hosts. The status programs display the results of their monitoring and command activities either as a series of reports or as a table, whichever is appropriate. A graphical interface that integrates most of these functions is also available on hosts with the required graphics hardware.

The information monitored by these programs includes network traffic statistics, host and manager status, and manager specific resource information such as available file space, cache hit rates and processor loading. In addition, manager transactions logs are recorded and accessible throughout the cluster using the Cronus file system.

#### 3.10.3.1 Host Probes and Service Probes

Host probes are supported on all Cronus hosts to reply to "are you there" requests. Service probes are similar monitoring entities in all Cronus services. The service probes are implemented as part of each manager. When a "report status" request is received by a particular manager process, the probe packages the current status and long term statistical data and transmits it to the requesting client. The operation switch also supports a "list services" request, to which it responds with a list of the currently active managers. Programs construct lists of available hosts and services by broadcasting an "are you there" request and examining the replies to determine the internet addresses of the active hosts.

### 3.10.3.2 Transaction Log

Each manager records a transaction log. A central trap log manager similarly records trap reports submitted by other managers. Any of the log files may be examined from any point in the Cronus cluster. Commands may be sent to the managers to change the degree of detail of the information recorded in the log files.

### 3.10.3.3 Status Display Programs

Commands exist for displaying the status of host, primal process, primal file, and directory objects, and for listing the active managers of a particular host. The replies are displayed in either textual or graphical format.

### 3.10.3.4 Starting and Stopping Services

Managers may be started and stopped by invoking an operation on the associated Cronus process. These requests are subject to access control based on the operator's access rights. Similar requests may be used to stop all Cronus manager activity on a particular host. When this command is received by the process manager, all Cronus processes will be terminated. The request may specify that the managers should not be restarted until instructions to do so are received; otherwise a specified set of managers will be immediately restarted. The program images needed to restart Cronus may be loaded from a remote location for hosts that cannot store the images locally. Some hosts are automatically restarted from the MCS if they fail to respond to active poll requests.

### 3.10.3.5 Graphical User Interface

We have also developed a graphical MCS user interface. This program provides integrated access to the entire collection of monitoring and control facilities. It replaces the individual commands and tabular data formats of the previous MCS implementation with icons and graphs. It also displays historical data, to indicate trends, and alerts the operator when certain situations occur, such as host or manager crashes.

### 3.10.3.5.1 Graphical Presentation

Economical graphics devices make a sophisticated MCS graphics interface practical. The MCS interface presents graphical, in addition to the traditional textual and tabular displays. The goal is to allow the user to view relationships between data values in a way that appeals to the users physical intuitions.

The simplest case consists using a gauge to display resource usage. The operator can recognize trouble when the meter reaches a certain position, rather than comparing two numbers representing the available and consumed resource amounts. A particular icon can simultaneously present several values, each encoded differently. For example, when metering the primal file service, we use the length of a bar gauge to represent percentage of file space occupied, a number on the face of the meter to display the actual values, and the brightness to indicate how fast retrieval requests are serviced. We make the gauge blink if the response time gets too long, thus drawing the operator's attention to the gauge. Historical data is shown on a graph, where the trend and the rate of change are apparent. Diagrams can be used to display relationships. For example, showing network traffic by the thickness of connecting lines between manager icons quickly gives the user a sense of where the bulk of the system activity is occurring and what pathways are relatively idle and could be either eliminated or other traffic rerouted to better divide the use. Thus, by effective use of graphics, we allow the MCS user to apply relatively quick visual perceptions, normally used to evaluate physical objects, rather than the slower analytical processes needed to evaluate tabular data or data presented on simple, unrelated graphical objects.

### 3.10.3.5.2 Interactive

Another goal for the MCS interface is that the operator should not need to remember complicated sets of options or names of specific hosts. When the operator is expected to enter a command, the full range of appropriate choices are presented; a mouse cursor may then be used to select the desired command. The operator can choose items by selecting a visible icon denoting the item.

We also seek to guide operator attention when unexpected events, such as host crashes, occur. For this we use multiple windows. A special message window is always visible. This is

used by the MCS to request action by the operator. If a report indicates operator attention is required, a notice is posted in the message window, with details provided in another window that the operator may select to review the situation. If the report specified which components were affected, a special menu will be provided for that report window that will allow the operator to quickly select views showing the affected components, with the affected components highlighted to help the operator locate them on the screen.

Our initial implementation provides high level views showing composite data, such as overall available file space, total active processes in the cluster, live host counts and network traffic. Exploded views are accessed by selecting the icons representing a service or host. Selecting a service icon gives a breakdown by host of the hosts supporting managers for that service. Selecting a host icon gives a breakdown of the managers running on that host. Selecting a manager icon, gives more detailed information on the selected manager. The user may also reduce the level of detail by using menu selections to return to summary views.

Icons for quick access to associated managers are provided. This reduces the need to step through several views when switching between service and host oriented views. Also, the same information may appear on several views. The MCS system continuously polls the cluster. It allows the user to examine various views of the system and to start and stop managers and hosts.

#### 3.10.3.6 Configuration Management

A configuration manager is used to provide a single, consistent view of the current system and of its configured resources. This information can be used to identify which managers participate in providing a particular service, to determine the hardware configurations for which a new software revision must be produced, and to format new views for the MCS operator.

### 3.10.3.7 Structure of the MCS

The MCS consists of several cooperating processes. These components may run on one or several hosts, and some components may be appropriately duplicated. It stores data using the Cronus file system. It uses the Cronus IPC to communicate among its components and with the services it monitors.

There are five functional components: the user interface, data routing, analysis, recording and retrieval, outbound command dispatching and poll regulation, configuration management, and service monitors and probes.

The MCS functions can be executed from anywhere in the cluster. Failure of the MCS or its operator does not endanger DOS survivability. To achieve this, the MCS follows three guidelines: key operator interactions can be accomplished from any access point (Cronus access control mechanisms distinguish Cronus operator requests from those of the non-operator user), the MCS functions are split into separate components which may be distributed and reliable, as appropriate, and Cronus object managers are designed to operate independently of the MCS, with the MCS providing instrumentation and control for the operator and nonessential advisory services to managers based upon its global system purview.

## 3.11 Resource Management in Cronus

### 3.11.1 General Approach

As a distributed system architecture, Cronus faces a number of resource management issues not present in non-distributed architectures. Strategies for effectively controlling the redundancy and configuration flexibility inherent in Cronus are needed to take advantage of the distributed system environment. These strategies for resource management are often conveniently separated into policies and mechanisms.

A policy is a goal or guideline set by a system administrator constraining the decisions made by a resource allocator. An intelligently formulated policy is based on an effort to maximize an overall benefit measure for the system. For example, a system-wide policy might be to evenly distribute resource utilization with the intent of minimizing the impact of a single system outage.

A mechanism is an internal system structure designed to implement a class of policies. For example, operating systems sometimes divide the processor manager into two components: a dispatcher or scheduler and a policy module. The dispatcher maintains a list of processes requesting the processor, sorted on a numeric priority field. Periodically, the dispatcher gains control of the processor, whereupon it activates the process with the highest priority on the list. The dispatcher thus implements a mechanism for priority scheduling, but does not determine a policy. The policy module, on the other hand, is responsible for periodically computing the priority of each requesting process, typically based on administratively-determined parameters, together with measurements obtained from the dispatcher. Varying the values of the parameters considered, or instructing the policy module to utilize additional parameters are both techniques for changing the resource management policy using the standard priority mechanism. The separation of policy and mechanism plays an important role in Cronus resource management.

In the Cronus system model, there are currently two general aspects of resource allocation which are particular to the network environment and must be effectively managed. One of these is the binding of a request from a client to a particular resource manager for those resources which are available redundantly. Redundancy comes in two forms: replicated objects (e.g., a multi-copy file) and replicated managers, any of which can create a new instance of an object type. In both cases the selection of an object manager to provide the given service is an important resource management decision. The other important aspect of resource management is the ability to dynamically migrate objects. This is a powerful tool for matching system resources to tasks in a manner that attempts to maximize some measure of system performance, reliability, or survivability. Both static reconfiguration (e.g., moving an entire collection of migratable objects at once), and dynamic reconfiguration (e.g., moving an individual object in direct response to demand for its use) are possible in the Cronus architecture and design.

The general approach to resource management in Cronus is to individually control the management of the classes of objects which make up the system. This approach extends Cronus resource management concepts to the abstract resources developed by applications. Resource management for an individual abstract resource (type) in Cronus is based on integrating a number of carefully planned mechanisms already in the system architecture. In addition to resource management by resource type, application

and system interface code can, if they choose to do so, control resource management decisions to incorporate larger purviews such as implementing an application specific policy which manages collections of object types used in a specific context.

In Cronus we achieve global and easily controllable resource management by requiring the object managers to cooperate in enforcing a resource management policy for their resource type. An object manager can redirect operations to a peer manager on another processor on the basis of current resource status. In the case of files, this means a file manager can redirect a request for creation of a file to an alternate file manager which may have more storage available. Part of the basis for decisions to redirect requests are parameters, settable dynamically by system administrators through monitoring and control functions, which control the resource management strategy. The creation of objects and resource management in general thus becomes a responsibility that is decentralized among object managers on each processor based on a global allocation policy, both in terms of sharing current status and possible redirection of operations between managers. The ultimate objective is to develop a type-independent model (and associated mechanisms) for resource management in Cronus, to be used off-the-shelf by future object type managers. This model would be capable of supporting a variety of policies.

The Cronus resource management model is based on the the integration of the following set of primitive mechanisms.

- o the ability of the kernel to route a request for a given type to any available manager of that type, using the Locate mechanism
- o the ability of a Cronus manager to redirect a request to a more appropriate peer to accomplish resource management objectives
- o the ability of managers to periodically accumulate current status of their peers via a standard mechanism (Report Status) to be used as a basis for selecting a site for redirecting an operation
- o The ability of users or applications to optionally indicate specific location preferences with requests
- o the ability to monitor and regulate the functioning of the

resource management policy from the monitoring and control station

- o the ability of applications to utilize system mechanisms (e.g. broadcast, Report Status operations) to build special-purpose resource management strategies tailored to their needs.

There is a hierarchy of desirable locations for implementing resource management policies: object managers, shared libraries, and finally application programs or users themselves. Managers are the most desirable because there are a limited number of them, they are readily identifiable and addressable, and they are most likely to be under administrative control (so policy parameters could easily be adjusted from the MCS). We anticipate that many decisions can be negotiated between managers based upon information periodically obtained via the generic operation ReportStatus. The Monitoring and Control Station also uses the ReportStatus operation to present the operator with a global view of how well the resource management policy is proceeding. Of course, these policies refer to automatic decisions made by the system; mechanisms for explicit user control of resource allocation continue to be available for those applications requiring it. Other application-dependent facilities could be built upon these explicit control mechanisms to implement user preferences and requirements.

### 3.11.2 A Resource Management Example

One aspect of resource management in Cronus involves a mechanism in which a manager of some given resource can, upon receipt of a service request, determine that it is not well suited to handle the given request and then make an attempt at finding another manager (of the same resource type) that is more likely to satisfactorily complete the client's request. More specifically, file manager resource management offers a flexible way to take advantage of a distributed filesystem and provide a mechanism to transparently improve file system performance for the client. Within Cronus, we have initially experimented with filesystem resource management for the file creation operation. In the future the same approach to resource management will be extended to cover other file operations and as well as other object types.



Before discussing how the various mechanisms are used in file system resource management, it is useful to review the basic steps in file creation within Cronus. A Cronus file can be created by either specifying the destination host on which to have the file created or with a generic create with which the system will locate a host for the client and send the request to that location. If the client has specified the host on which the file is to be created then that host's file manager will either service the request or, if there are circumstances which prevent it from doing so, it will reject the request and send a negative acknowledgement to the client. Alternatively, if the client has not specified the target host for file creation then any host is free to answer the request from the client's operation switch request to locate a file manager. When a file manager has answered a locate, the client's original create request is forwarded by the operation switch to that manager. It is useful to note that without resource management, answering the locate request implicitly allows that manager to be selected for the file creation without any regard for its ability to handle the request with any degree of efficiency or reliability.

The basic elements needed for performing resource management for Cronus files lie mostly in the file managers themselves. Upon receiving a file creation request, the manager must be able to analyze what its current capabilities are for successfully providing that service. The method used for calculating what the 'health' of a given file manager is at any given moment is through an objective function. This function inputs the values of several variables kept by the manager and returns a single value which will tell the file manager if it should go ahead and perform the file creation or take alternative action. Currently the variables maintained for the objective function to use as inputs are.

#### Filesystem Capacity

This variable relates to the percentage of available space in the the Cronus filesystem on that host. If this percentage ever falls below a preselected limit then the objective function will return a value that would indicate this file manager unfit (except under emergency conditions) for file creation regardless of the values for other variables. This file manager would still service other types of requests (such as reading, deleting and reportstatus operations) since these operations don't require additional disk storage.

### Host Load Average

The load on the host on which the file manager resides is tested to see if the level appears high enough to prevent the manager from expediently completing the requested operation. In the event of a heavy load, a file manager is still capable of successfully carrying out the operation but performance considerations make it desirable to find a better candidate to service the client.

### Objective Variable

An objective variable is included to provide to the operator a 'tunable' parameter for affecting how managers will accept or reject operations based on their objective function values. This variable is intended to be set from an operator at the MCS intimately familiar with the entire Cronus cluster. If the operator decides that a particular file manager is in fine shape then this variable should be set to a high value. If for any reason the operator decides that some file manager should not bear much of the systemwide file creation load (or none at all) then this value is set low (or to zero to keep it from performing any creations at all).

If the objective function returns a value that tells the file manager not to perform the operation itself the manager attempts to forward the request to a 'healthier' file manager. Obtaining the status of alternate sites for handling this request is done by broadcasting to all other peer file managers the generic operation reportstatus. Included in the reportstatus reply are parameters which are used to compute the objective function for other managers. The manager computes objective functions for at least some other peer managers, and selects one with a better objective function. If it can't find a manager whose status is better than it's own it can do the operation itself, if at all possible, or else it may be forced to reject the client's request. If a candidate has been found to assume the file creation request then the original creation request is forwarded to that host and the responsibility for servicing the client falls to that host's file manager.

We are currently experimenting with these mechanisms and tunable parameters before applying the general resource management strategy to other resources and attempting to incorporate it into the automated manager development software.

#### 4 Test and Evaluation

The testing and evaluation of new technology is a difficult undertaking. Best results seem to emerge when a new technology or product is made available to potential users early in the development cycle. However, on the other side of this issue, early models of a technology or product are most likely to be very flawed and incomplete, and likely to dissuade any serious use except by really committed organizations. Without some degree of serious use there can not be any serious test and evaluation. In the case of Cronus in particular, because it is a large and evolving system, early use by an outside user community is just not feasible. But precisely because it is large and evolving, early use can help identify areas of deficiency at a time when they are easier to identify and correct.

To deal with this apparent contradiction we have focussed our initial system test and evaluation activities on our own use of the system as its parts become available. In some cases this implies nothing more than a standard, "layered" bottom up development process, where new levels of functionality make direct use as clients of interfaces developed for the lower layers. In other cases, this implies additional tasks and effort to tailor emerging system functionality toward areas which would be of immediate use to us in our role as software designers, developers, and maintainers. In the rest of this section we discuss both aspects of our approach to test and evaluation, and make a few preliminary conclusions of results to date.

##### 4.1 Areas of Internal Use

There are three major aspects of our internal use of the emerging Cronus system and software. These are:

- o Applying the basic Cronus decomposition and distribution methodology to the development of the system itself.
- o Using network support software on testbed hosts to provide initial access to the cluster machines for purposes of software development.
- o Constructing specific Cronus services and integrating specific software development tools to allow the use of Cronus components as an aid in developing and supporting

additional Cronus software development.

By applying the Cronus decomposition methodology we mean that major parts of the system itself and many of its intended applications are structured as interactions among and between abstract objects distributed throughout the system and clients requesting the manipulation of these abstract objects. Files, processes, principals, directories, etc. are system objects (i.e. supplied along with the basic system) which are on equal footing with application specific objects and object managers. Thus, when we are developing catalog managers, authentication managers and support for other system objects we are utilizing internal interfaces and relying on the correct functioning of the same support software as will the future application developer. In addition, some system components make direct use of others in their implementation. As examples, the catalog manager uses Cronus primal files for storing directory objects, and the "names" of access control groups are cataloged in the Cronus catalog. These implementations provide early and heavy use of many system components.

Enhancements and refinements made to the object system support code as a result of initial use in developing parts of the system's intended functionality has resulted both in better performance of these functions and better support for future distributed applications. Our approach is to initially experiment with "hand coded" implementations to gain experience with the real issues and problems which will face the programmer before developing high level tools for future Cronus application programmers. The first few Cronus object managers were coded by manually inserting calls to support library routines, while the last few have benefited from this experience and the subsequent automation of a number of steps in the development procedure.

Our testbed environment consists of a number of different types of systems. Networking these machines together via the local area network was an early project effort. The current design of Cronus requires as a support component implementations of the IP/TCP protocols, in addition to local network support, for each participating host. For some hosts, these implementations already existed and came with the system, for others they needed to be procured from a separate vendor, for still others we had to design and build our own. In each case, the software needed to support Cronus IPC was largely untested especially under load conditions.

As a means of both improving the accessibility of our machines to development staff and to evaluate the adequacy and reliability of the network support code, we began by developing support for standard network protocols such as Telnet and FTP to stimulate daily use of the newly developed or acquired software. In addition, to ascertain the adequacy of network performance, extensive measurements of Ethernet, IP and TCP response time and throughput were made. These were reported in Cronus Interim Technical Report No. 2 (BBN Report No. 5261).

While the evaluation approach was very positive, the results were mixed. One implementation of network software (C/70 UNIX) proved buggy but fixable, one implementation (M68000 CMOS GCE) proved to be fairly reliable after some initial problems were cleared up, and one implementation (Compton software for VAX-VMS) proved very unreliable and problem prone. It is scheduled to be replaced by another implementation as soon as possible.

The final aspect of current test and evaluation was to orient initial use of the system toward software development application programs. This too has a dual purpose. First, it will exercise additional Cronus components on a daily basis, including file manager, catalog manager and user authentication. Second, the software tools and applications developed for or integrated into Cronus establish a starting point for providing future application developers a set of software development tools which support more convenient use of resources distributed throughout the cluster. Initial use involved the development of a distributed file system and integration of standard word processing and basic software production (compiler etc.) tools which can operate in the distributed file environment. Early experience has led to a number of improvements, especially in performance, of the basic system functionality, as well as the recognition of a number of areas requiring additional effort. Three of these important areas identified as requiring additional work in the next phase of the effort include the survivability of key system functions (e.g. catalog survivability), continued performance improvement and improved tools for distributing new versions of system components to the many and varied hosts of the configuration.

Using the system as a basis for a continuing program of test and evaluation has proven extremely valuable in validating approaches and uncovering problems, although it is becoming more costly as we become dependent on more of the system being

available daily. This has resulted in devoting more project resources to system maintenance activities, and consequently less to new development.

*MISSION*  
*of*  
*Rome Air Development Center*

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C<sup>3</sup>I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C<sup>3</sup>I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.