(4)

RADC-TR-88-81
Final Technical Report
April 1988

CRONUS, A DISTRIBUTED OPERATING SYSTEM: REVISED SYSTEM/SUBSYSTEM SPECIFICATION

BBN Laboratories Incorporated

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

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

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.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED: Raymond P. Urtz Jr.

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

FOR THE COMMANDER: John A. Ritz

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) 5884	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-81
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: REVISED SYSTEM/SUBSYSTEM SPECIFICATION

12. PERSONAL AUTHOR(S) Richard W. Schantz, Robert H. Thomas, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, R. Sands and K. Lebowitz

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Sep 81 TO Sep 84	April 1988	250

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; System Monitoring and Control; Interoperability; Survivable Application; Heterogeneous Distributed System

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
This report presents the current design for CRONUS, the system being developed under the Distributed Operating System Design and Implementation project sponsored by Rome Air Development Center. It is intended as an overview of the system structure and as a synopsis of the current system/subsystem decomposition and specification.

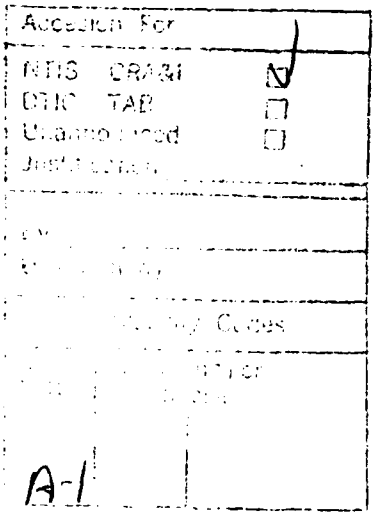
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas F. Lawrence	22b. TELEPHONE (Include Area Code) (315) 330-2158 22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Table of Contents

FIGURES



TABLES

1 Introduction

This report presents the current design for Cronus, the system being developed under the Distributed Operating System Design and Implementation project sponsored by Rome Air Development Center(1). It is intended as an overview of the system structure and as a synopsis of the current system/subsystem decomposition and specification.

This report is a revision to two earlier drafts, BBN Report No. 5260, November 1982, and BBN Report No. 5646, May 1984. A previous report, "Cronus, A Distributed Operating System. Functional Definition and System Concept", BBN Report No. 5884 is intended as a companion to the current report, and the reader is assumed to be familiar with its contents.

In Section 2, we briefly review a few of the areas covered in the Functional Definition, and extend them to cover current development plans.

Section 3 presents an overview of the Cronus operating system, stressing the common framework into which its components will fit and the functional decomposition of the system.

Sections 4 through 12 present the design for the various system functions. In a number of areas the design is only partially complete. These sections will form the basis of a continuing and evolving subsystem specification for the various components, throughout the life of the project.

Section 13 sketches how the system supports some common functions. Other Sections contain a description of the system environment, including hardware, Virtual Local Network, GCE software, and system utilities and libraries.

---

(1). This work is being performed under RADC contract No. F30602-81-C-0132

In previous versions of this document, detailed descriptions of the commands and functions in this system, as well as of the objects, operations, and formats used in the decomposition, were included. Much of this material is more appropriate to the Cronus User's Manual. Many details which were included in the earlier versions of the System/Subsystem Description have been removed from this report to the User's Manual. In addition, detailed design notes made during the implemention of the system are included there. Cross references to this document appear throughout the System/Subsystem Description. These are of the form (see Cronus User's Manual topic(number)); where topic is the name of a page in the manual, and number is the section number within the Cronus User's Manual where one may find the page.

Many people, in addition to the current Cronus project development staff listed as authors of this report, have contributed both ideas and enthusiastic effort in designing and constructing the system described. These people include William MacGregor, Benjamin Woznick, David Mankins, Robert Walsh, Ed Burke, Steve Toner, Mort Hoffman and Steve Geyer.

2 Cronus Project Overview

2.1 Project Objectives

The objective of the Cronus project is to develop a testbed for evaluating distributed system technology. To do this we are establishing a prototype local area network based hardware architecture, and building an operating system and software architecture to organize and control this distributed system. The architecture was partially specified by the statement of work, and further defined during early stages of the project. It is described in the Cronus Functional Description [BBN 5041], and is summarized in Section 2.4. In addition to establishing a system architecture, the other major aspects of the Cronus project activities are:

1) Select off-the-shelf hardware and software components as a basis for an Advanced Development Model (ADM) prototype configuration for the distributed system testbed;

2) Design the system;

3) Implement a version of the basic system components;

4) Test and evaluate the concepts and realization of the DOS in the Advanced Development Model.

The orientation we have chosen is both experimental through construction of working system components, and evolutionary through pre-planned continuation of design and development activities.

2.2 Points of Emphasis

The Cronus design is intended to introduce a coherence and uniformity to a set of otherwise independent and disjoint computer systems. This grouping of machines, operating under the control of a distributed operating system, is called a Cronus cluster. The aim is to provide for the cluster configuration as a whole features comparable to those found in a modern centralized computer utility. There are various ways of viewing

this uniformity and coherence, each plays a role in the Cronus design.

From an end user's point of view, the Cronus DOS provides a single account with access to all integrated system services, a uniform distributed filing system and a uniform program execution facility, which is independent of the site of the activity. From a programmer's point of view, Cronus provides a uniform interface and access path to the distributed system resources, and supports the initiation and control of distributed computations. More importantly, from both an end user's and programmer's perspective, Cronus provides a common system framework for applications. This means that otherwise independent computerized activities can be constructed so that they are more easily made to work together, despite implementations which cross host and processor-type boundaries.

From an operations and administrative perspective Cronus provides a logically centralized facility for monitoring and controlling all of the connected systems. Functions such as account authorization, user priority, and access control can be applied system-wide rather than individually to each host.

In addition to coherence and uniformity, there are a number of other system design goals. These are:

- Survivability and integrity of Cronus itself,
- Scalability to accommodate both small and large configurations,
- Experimentation with resource management strategies that effect global performance,
- Component substitutability to allow easy use of alternate functionally equivalent hardware, and
- Convenient operation and maintenance procedures.

2.3 System Phases

System development consists of three phases. The first phase, coincident with the development of the functional definition, included component selection, installation, interconnection and testing. The second phase includes the design and implementation of the basic system that will provide the uniformity and coherency to the collection of machines. It also provides the framework for the in-depth design, implementation, and experimentation in the other areas of interest (e.g. survivability), which are to occur as the third phase. The second phase design is the principal subject of the remaining sections of this report. In certain areas, elements the third phase design are sketched as well.

2.4 The Cronus Hardware Architecture

2.4.1 System Environment

The Cronus environment consists of several parts: the local area network which provides the communications substrate for a Cronus cluster, the set of hosts upon which the Cronus system operates, and a mechanism for connecting a Cronus cluster to the Internet environment and to other Cronus clusters.

Cronus enables a variety of constituent computer systems to operate in an integrated manner. Cronus is distinguished from other distributed operating systems by one or more of the following characteristics:

1. Cronus will run on a group of heterogeneous hosts.

2. Cronus hosts will run operating systems which are largely unmodified. The Cronus distributed operating system software runs as an adjunct rather than a replacement for the hosts' primary operating systems.

3. Hosts will be included in Cronus with varying degrees of system integration. Some support limited subsets of the services defined by the Cronus environment.

4. The interconnection network is designed on a hierarchical model. A Cronus cluster includes a set of hosts connected by a high-speed, low-latency local network. A set of Cronus clusters may be connected over slower long-haul networks.

The Cronus architecture provides a flexible environment for connecting hosts so that facilities available on one host may be conveniently used from other hosts. It provides two alternative host integration schemes. A host may implement the Cronus Interprocess Communication (IPC) mechanism and have efficient communication and operations with the rest of the Cronus hosts, or it may access the other Cronus hosts through a front end access machine, which is a simpler, less expensive option for connection of a host, but which may be more limited from a flexibility and performance viewpoint.

2.4.2 Host Classes

Cronus hosts can be divided into four groups, mainframe hosts, Generic Computing Elements (GCEs), workstations, and internet gateways.

The collection of mainframe hosts, each of which serves a number of users simultaneously, includes a variety of machines with unrelated architecture. A mainframe host may be tightly integrated into the system, both offering and using Cronus services and fully implementing Cronus interprocess communication. Alternatively, they may be loosely integrated, offering no services, possibly connecting into Cronus through an access machine which provides communication with the rest of Cronus.

GCEs are small, dedicated-function microprocessor based computers of a single architecture but varying configuration. Each GCE provides a basic service. For example, a GCE can be a file manager, a terminal manager, an access machine or it might carry out a more complex system function as an authorization manager. Since all GCEs have the same architecture, they provide a replicated resource which, with the appropriate software, enhances the reliability of basic Cronus functions.

Workstations are powerful, dedicated computers which provide substantial computing power and graphics capability at the disposal of a single user. They differ from mainframes in that they support a single user. They differ from terminals in that they offer significant computational resources.

An internet gateway is a computer used to interface communication between multiple networks. The Cronus gateway integrates the Cronus cluster into the collection of networks known as the ARPA internet and provides a base for supporting remote access and intercluster communication.

2.4.3 System Access

There are a variety of user access paths to Cronus. One is a connection by means of a Cronus terminal concentrator. Users may gain access through the internet gateway from remote points. Cronus also supports access through terminal access mechanisms on its mainframe hosts. These latter two access paths provide the same interface to the user as the terminal concentrator. Access from a workstation may be different than from a terminal, since the workstation defines the user interface. The user has immediate access to the workstation's capabilities.

2.4.4 Local Area Network

The set of hosts is connected by a local area network. The characteristics of the network are crucial to the success of Cronus, since they determine the kinds of communication and operations that are feasible across host components of Cronus.

The selection of an Ethernet for the local area network for the Advanced Development Model has been described in a recent report [BBN 5086]. This choice was motivated by criteria in the project's statement of work:

1. The network should be suitable to support a distributed

operating system.

2. The network should be currently available and economical. Since the Advanced Development Model will not be operated in a stressed environment, certain constraints applicable to a field-deployable version were considerably relaxed.

The Ethernet was chosen for the local area network substrate for the following reasons.

- o The network must be "high-speed". For the ADM, the network should operate at rates of Megabits per second (MBits) with low latency, with higher speeds desirable. The Ethernet operates at 10 MBits.

- o Network interfaces to all or most of the computer systems in the DOS ADM should be available. With the exception of the C70, whose Ethernet interface has been constructed under the present contract, this was the case.

- o The local network must provide a datagram-style service.

The Ethernet fulfills all three requirements and we believe is, at the present time, the most cost-effective network technology which does. In addition, the Ethernet provides broadcast and multicast capabilities which, have been extensively exploited in the system design.

The raw Ethernet layer will not be used directly. To achieve convenient substitutability of alternate communication substrates, Cronus will use an abstraction of the Ethernet capabilities which is provided by a Virtual Local Net (VLN) software layer, described in Section 14.2. The VLN represents an enhancement of the DOD standard IP protocol to provide for features common to local area communication. We anticipate that future versions of Cronus will need to be built upon a different local network, such as the Flexible Interconnect, which have reliability, communication security, and ruggedization not available in current commercial products. By designing the VLN layer and building Cronus upon it, it should be easy to substitute any local network that provides the basic transport services required by Cronus.

2.4.5 Types of Hosts

GCEs are implemented in the ADM system by Multibus computers with Sun processor board (the current vendor, one of several, is Forward Technology) processors, large main memories, an Ethernet controller, and additional hardware (disks, RS-232 ports, etc) needed to support specific functions(2). The Multibus computers were chosen because

1. They are relatively inexpensive, permitting low cost incremental system growth.

2. The Multibus standard guarantees the ability to package individual GCEs in different ways with components from a variety of vendors.

3. New processors and devices are expected to evolve for the Multibus over time.

Utility hosts provide the program development and application execution environments for Cronus. In the ADM, this function will be supported by C70 UNIX systems, VAX-UNIX Systems and a VAX-VMS System. UNIX was chosen due to the rich set of development tools already available for it and the ease of developing new tools and applications. The C70 was chosen because it was one of the least expensive computers which supports a multi-user UNIX, and because of the in-house expertise and support for the hardware base. The UNIX support will be gradually shifting to VAX-UNIX. A VAX running the VMS operating system was chosen to demonstrate the handling of heterogeneous systems.

---

(2). One of the functions we would normally install on a GCE is the Cronus Internet Gateway, which will be installed on an DEC LSI-11 computer instead, because the standard Internet Gateway implementation uses the LSI-11.

2.4.6 Cronus Clusters and the Internet

The goal of the Cronus project is development of a local area network-based distributed operating system. The Cronus cluster will operate in the Internet environment as a class B network. Cronus hosts will support the DoD Internet Protocol (IP) for datagram traffic, and, where connections are required, the DoD Transmission Control Protocol (TCP).

A Cronus cluster is expected to use the Internet environment in two ways. First, access will be provided to Cronus from points in the Internet external to the cluster. Second, the Internet will support communication between distinct Cronus clusters.

2.4.7 The Advanced Development Model

The Advanced Development Model (ADM) of Cronus is the first instantiation of the Cronus hardware and software. It is, as its name suggests, the development testbed for Cronus. The ADM is experimental and can be expected to undergo rapid change as Cronus is developed, software is implemented, altered, and improved.

The ADM is being assembled using many off-the-shelf commercial hardware and software component building blocks. This reduces the cost of its components, permits the use of newly available state-of-the-art hardware, and enables us to be more flexible in its design. We are developing a design with the sufficient flexibility to permit later substitution of more suitable hardware and software for deployable configurations.

3 System Overview

A distributed operating system manages the resources of a collection of connected computers and defines functions and interfaces available to application programs on system hosts. Cronus provides functions and interfaces similar to those found in any modern, interactive operating system (see the Cronus Functional Definition and System Concept Report [BBN 5041]). Cronus functions, however, are not limited in scope to a single host. Both the invocation of a function and its effects may cross host boundaries. The distributed functions which Cronus supports are.

- o generalized object management
- o global name management
- o authentication and access control
- o process and user session management
- o interprocess communication
- o a distributed file system
- o input/output processing
- o system access
- o user interface
- o system monitoring and control.

In this section, we introduce the Cronus design and briefly discuss the major elements of the system decomposition.

3.1 System Concept

The primary design goal for Cronus is to provide a uniformity and coherence to its system functions throughout the cluster. Host-independent, uniform access to data and services forms the cornerstone for resource sharing. The design of Cronus is based on an abstract object model. In this model, we treat the system as a collection of objects organized in classes. files, processes, directories, and so forth. Only a limited number of well-defined operations can be invoked on an object, and the only information that a client can have about the structure or content of the object is obtained through these operations. The system structure is defined by the objects which consitute the system, the operations on these objects, and the responses which the objects give to the operations. The

underlying structure of the system, which is essentially hidden from the clients, consists of the primitives which deliver the operations to active objects (processes), or to processes which are responsible for passive objects like files.

The Cronus distributed operating system is built from a number of concurrently existing objects called processes that reside on hosts which are part of the cluster. Some of them, called object managers, play a special role in implementing other objects of the system. Other processes provide services and functions for the clients of the system. Still other processes run user programs. Processes communicate with each other to form larger abstractions and build more complex objects. At the most fundamental level, communication between processes is through messages sent over a local area network connecting the hosts of the cluster.

There are four interrelated parts to the Cronus system model.

- A 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. This part of the system includes facilities for locating an object and controlling access to it.

- A set of basic object types, along with the object managers which implement them. There are two groups of basic object types. One group is fundamental to the development of new object managers in Cronus. This group of object types includes processes, user records and symbolic name directories. Another group of basic objects is provided to support various application domains and processing requirements. Initially for Cronus this includes files and I/O devices.

- A paradigm for building and accessing new types of objects, which spells out the methods for integrating new object managers.

- User interfaces and related utility programs to provide convenient access for both people and programs to the system objects and services.

3.2 The Cronus Object Model

The object model provides a coherent and uniform framework for the system components of Cronus, and potentially for application programs in a Cronus cluster. Since a distributed operating system is itself a distributed application, the methodology used in its construction should apply equally well to the construction of other distributed applications. The references [Xerox 1981, Rentsch 1982] discuss the object-oriented model of programming. The following are the key features of the object-oriented model that Cronus supports:

- Each Cronus object is a member of a well-defined class, which is called the type of the object. The names of Cronus types begin with the string 'CT_'. a list of some of the more important types may be found in Table 1.

- There is a set of operations (often called methods in the literature) defined for each Cronus type. These define the only ways that an object can be examined or modified.

- Every Cronus object has a unique identifier (UZD) name. References to the object are generally through its UID, which is a bitstring uniquely identifying the object over the entire Cronus cluster. Cronus also has a symbolic catalog for cataloging UID's to provide convenient reference to objects.

- The primitive Invoke causes a named operation to be performed on a named object.

- There is a basic set of operations (called generic operations) which are defined for all objects; these operations promote a unity among the various object types of the system and constitutes a limited form of inheritance of the operations defined for the basic type CT_Object. These operations include those which create and remove objects, and those which control access. Each Cronus type then has its own operations, and may redefine operations which are known to its parent class.

- An object has one or more parts that are visible to the outside world. These may include data, an object descriptor, and an active (or process) component. All

Cronus objects have at least an object descriptor, which is the repository for such information as access rights.

Object Name	See Section
CT_Object	4.2
CT_Host	5.1.4
CT_Cronus_Process	5.1.2
CT_Primal_Process	5.1.3
CT_Program_Carrier	5.2
CT_Cronus_Catalog	9.2
CT_Catalog_Entry	9.2.1
CT_Directory	9.2.2
CT_Symbolic_Link	9.2.3
CT_External_Link	9.2.4
CT_Cronus_File	8.1
CT_Primal_File	8.1
CT_Migratory_File	8.1
CT_Dispersed_File	8.1
CT_Executable_File	8.1
CT_Principal	7.5.2
CT_Group	7.5.3
CT_Authentication_Data	7.5.1
CT_Session_Data	11
CT_Line_Printer	10

Table 3.1 Cronus Objects

Fundamentally, the implementation of the Cronus system kernel consists of the implementation of the primitive Invoke. Each object is associated with an object manager, which knows all the internal details of the construction and location of the object.

When an operation is invoked on an object, the Cronus kernel is responsible for delivering the operation to the appropriate object manager, which performs the task requested in the operation, and, if appropriate, responds to the invoker.

The operation switch in the Cronus kernel supports both invocations of operations on objects and message communication between processes. Since processes are system objects with defined operations to send and receive messages, the operation switch provides a host-independent interprocess communication (IPC) facility for both the system implementation and application programs. Further details of the object model and the design of the operation switch are described in Section 4.

Some of the attractiveness of a distributed architecture is the potential to utilize redundancy and configuration flexibility interest in the hardware architecture. Cronus supports a unified approach to these attributes through its object orientation. In general, three somewhat different classes of objects will be accessed in Cronus. These are.

1. Primal Objects

   These are forever bound to the host that created them. There is no simpler form of Cronus object. An example would be a Primal File, which is permanently bound to its storage site.

2. Migratory Objects

   These are objects that may move from host to host as situations and configurations change. A standard Cronus mechanism can locate the current site to complete an object access.

3. Structured and Replicated Objects

   These are objects which have more internal structure than a single uniquely identified object. For example, a replicated file would have a number of primal files as its constituent parts. The UID would be recognized by manager processes on each of the sites for the more primitive elements. Replicated objects are a key element in Cronus system survivability.

Cronus can be extended by adding new object types to support new requirements or functions. Certain features are required for each object type including supporting the generic operations. In addition, the object model and its associated system components define a number of system conventions such as, integration with the monitoring and control software which may be adopted by subsystem designers, on a case-by-case basis. A subsystem designer can depend upon the existence of required features in other system components, and is obligated to provide them in each new component. On the other hand, the Cronus system design minimizes the number of required features for system entities, which, in turn, reduces the buy-in costs for new hosts and object types.

Maintaining the integrity of complex objects is the responsibility of the managers for the type. This means that techniques can be tailored to the patterns of access to the object being maintained.

Since the generic operations include those which manage access permissions, uniform access control is a basic part of the Cronus object model. The object managers control access to the objects they maintain through the use of access control lists (ACL). The operation switch reliably stamps the UID of the invoking process on each of its messages, so the process making the request can be reliably identified.

The conventions for communication are based on the message structure library (MSL). A message consists of key-value pairs. There are also conventions that provide simple transaction protocols, and other features to support flexible message handling and processing. The MSL also standardizes the representation of data types, which allows the common interpretation of data items across a Cronus cluster. The MSL design is discussed in Section 6.

3.3 System Objects

To provide the initial operating capability, a number of basic system object types and their managers are being developed to support the functions outlined in the Cronus Functional Definition [BBN 5041]. They include.

- Process objects and process managers that support the Cronus system and user programmable processes. They may be linked together across the cluster, and connected through interprocess communication to form a user session.

- User identity objects and a permanent user data base that support authentication and access control.

- Directory objects and catalog managers that implement the global symbolic name space.

- File objects and file managers that provide a distributed filing system which can be used in providing non-volatile storage for developing portable object managers, as well as for satisfying application program data storage requirements.

- Device objects and device managers that support the integration of I/O devices into Cronus.

Much of the Cronus design has been decomposed into the subproblems of developing the Cronus distributed object model and of designing the components which provide these basic system objects. The design of these components is described in detail in Sections 4-12 and in the Cronus User's Manual.

3.4 Cronus Name Spaces and Catalogs

Cronus has two system-wide name spaces for referencing objects. The unique identifier (UID) for an object is the basic name. Unique identifiers are fixed-length, numeric quantities, intended for use by programs but unsuitable for people to read, remember, and type. The unique identifier has internal structure which Cronus uses, but is normally invisible to applications. It contains the name of object's type and the name of the host that generated it. The host name is useful as a hint for locating certain objects which do not migrate.

The Cronus system also includes a global symbolic name space oriented toward human use. Normally, the accessing agent would interact with the Cronus symbolic catalog manager to look up the

unique identifier for the object. After it obtains the UID, the accessing agent can then invoke operations on the object.

Although there is no single identifiable catalog supporting the UID name space, the notion of a catalog for UIDs is a useful abstraction. This catalog will be referred to as the UID Table; in practice, the functions that it supports are implemented by object managers for different object types by means of UID-to-object-descriptor tables, which can be thought of as fragments of the UID Table. When a Cronus object is assigned a UID an entry is created in a UID table. This entry contains the information that the manager needs to access the object. Object managers support two kinds of operations. The generic operations, for example, those used to create or remove an object, to modify the access control list, and to examine the object descriptor, are defined for all objects. Other operations may be defined only on a particular type, these are often called type-dependent operations.

The Cronus operation switch provides client processes with addressing based on the UID, so if a client process has the UID, it can communicate with the object. The UID is a universal name that can be used from any one of the hosts in the cluster to refer to the object, no matter where in the cluster it is stored. Although it may not happen often in practice, objects may move (or be moved) from one host to another. When an object is relocated in this fashion, its UID does not change. A replicated object also has a single, unique identifier for client access to any of its images. Replicated objects may be developed out of more primitive, non-replicated objects which are usually accessed directly only by the replicated object manager.

A Cronus unique identifier actually consists of a pair

<UNO, Type>

where UNO is an 80-bit unique number, and Type is a 16-bit value naming the type of the object. The UNO portion of the UID is uniquely associated with a particular object. Each Cronus service is assigned a type. In the current design, all types are statically well-known. Since the type field can encode as many as 65,536 distinct types, there is room for expansion to dynamic types at a later time.

Each Cronus type has a generic name associated with it, this is a UID that has the type portion set to the type of the object and UNO portion set to zero. Cronus generic names are used for a variety of purposes. They act as class names in many of the places one would expect, particularly when an object is being created. That is, the creation of an instance of a class is treated as an operation on the generic name. In addition, the generic name is used when the system is interrogating the operation switch to find a manager for the type. In the current implementation, the manager itself is implemented from a Cronus primal process, which has a UID of type CT_Primal_Process that was selected when the process was created. The operation switch is responsible for identifying the process that manages objects of a particular type. It does this by examining the type portion of the UID name on which the operation has been invoked.

The facility that generates unique numbers may be regarded as existing continuously throughout the life of a Cronus configuration, and is accessible to system and application processes. No two requests by client processes for a UNO ever obtain the same UNO. Hence the unique number generator is an example of a survivable distributed program. The generator must be survivable, because UIDs must be unique over the lifetime of the cluster, and it must be distributed, because without it new objects cannot be created, so it cannot depend on any single host being up.

The UNO consists of three fields, a HostNumber, a HostIncarnation and a SequenceNumber. The HostNumber is the Internet address of the host that generated the UNO. The SequenceNumber is incremented for each request. The HostIncarnation is incremented if the SequenceNumber overflows its field. It is also incremented whenever a host that has been down comes up. In order to assure the uniqueness of the UNOs which are generated, the HostIncarnation is kept in stable storage, either on the host itself or on some other host that supports stable storage.

The UNO size, 80 bits, was derived from assumptions about the number of UNOs that could be generated over the lifetime of the Cronus implementataion and the mean rate at which systems enter or and leave a cluster. The current field sizes will allow a mean generation rate of about 10,000 UNOs per host per second and a mean crash rate of once every 3 minutes for 100 years.

these numbers are assumed to be adequate for reasonable system activities.

The principal design consideration for the symbolic name space is to make it easy for people to use. Names for Cronus objects are uniform and host independent. Symbolic names are supported by a catalog that provides a mapping between symbolic names and the UIDs. This name space is a tree, composed of nodes and directed labeled arcs. There is a node called the root. 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 generally represent Cronus objects which have symbolic names. A complete symbolic name begins with the punctuation mark colon (:), followed by the names of the arcs, separated by colons. For example, :a:b:c is the symbolic name of an object.

Not all Cronus objects have symbolic names, and those that do may have more than one. When an object is given a symbolic name, an entry 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 supports Enter, Lookup, and Remove operations. In addition, operations are provided to read and to modify the contents of catalog entries.

The catalog is distributed, different hosts manage different parts of the name space. The implementation is logically integrated, however, so that any catalog manager process can be asked to perform any of the catalog operations. The upper portion of the hierarchy is replicated to support efficient access to different parts of the name space. The symbolic catalog is implemented out of more primitive directory objects, which adhere to the general Cronus object paradigm. The Cronus catalog is described in detail in Section 9.

3.5 The Cronus File System

The collection of all Cronus files constitutes the Cronus distributed file system. Within this file system, Cronus supports several file types. The most basic file is a primal file, which is stored entirely within a single host and is bound to that host throughout its lifetime. Other types of Cronus

files are built from primal files. A replicated (or multi-copy) file, which has multiple instances replicated across Cronus hosts for increased availability or enhanced responsiveness, is constructed from several primal files. Therefore, if a host contributes storage resources to Cronus, it must support primal files.

There is no single table that lists all file objects. Rather, each file manager owns all of the data for the file objects it manages. The Cronus object addressing facilities make possible a client interface in which knowledge of a UID is sufficient to access the file regardless of its location. Clients may make file placement decisions themselves if they wish. Alternatively, placement decisions will be made automatically.

Ordinary read and write operations may be performed on file objects. The expected mode of access to Cronus files is to transfer the file data as needed, much like conventional filesystem access to disk files. Copies of Cronus files are made only to satisfy explicit user requests or to support other system requirements. The design for the Cronus File System can be found in Section 8.

3.6 Cronus Process Management

There is more than one type of process object in Cronus. Primal processes are the simplest process entities. They are constructed from the process abstraction that exists in the constituent host operating system. This simple form of process is used as a building block for the system implementation, minimizing integration costs for new Cronus host types. Since primal processes cannot be loaded dynamically with user programs and lack flexible process control functions, they are too inflexible to be used as vehicles for general application programming, but are used as object managers and in other well-defined system roles.

To satisfy the requirements of application programs, primal processes are augmented with a subtype, the program carrier process, which supports a richer process environment. Program carrier processes can be loaded remotely and started in a manner

that is uniform across the cluster. In addition, program carriers support, in a host-independent manner, the kind of flexible control and interconnection of related processes found in modern operating systems.

Cronus processes have most of the features natural to the host on which they are built, and no attempt is made to hide these features. An application builder has the choice of when to use locally-supported features and when to use standardized Cronus features. To the extent that applications choose to adopt Cronus process features, they will be better integrated with the other cluster processing activities. On the other hand, the judicious use of local features will enhance the efficiency of the activity. Cronus processes are described in Section 5.

3.7 Device Integration

Special purpose devices, such as line printers and tape drive devices are important elements in a system configuration. As Cronus objects, these devices are available to the entire cluster through an object manager. In some cases, more elaborate interfaces can provide an access path with specialized features. For example, a line printer service, can be provided that supports spooling. Device integration is discussed in Section 10.

3.8 User Identities and Access Control

Users are represented by system objects, known as principals. A principal object contains data that describes the manner in which the user may use the system. This information supports operations such as authentication and session initialization. The object manager for the principal objects and for other access-related objects is called the Authentication Manager. The Authentication Manager component services the entire cluster.

The purpose of Cronus access control is to prevent unauthorized access to Cronus objects. This is done uniformly by associating an access control list (ACL) with each object.

Access is then either granted or denied based on the identity of the principal associated with the accessing agent and the contents of the access control list for the object.

The operations of the Authorization Manager and the access control system are discussed in Section 7.

3.9 Process Support Library

The Process Support Library (PSL) is a collection of functions, that may be bound into the load image of a Cronus process.

PSL routines are considered part of the Cronus system and are generally supplied with the system and maintained by system programmers. The PSL fills the following major roles.

1. It provides a convenient interface to Cronus operations.

2. It provides access to special Cronus features such as the facilities which generate UNOs and structure messages, and to the elementary file system that underlies the primal file sytem; It also provides a uniform interface to the interprocess communication facility. These features are not normally accessed though the Operation Switch.

3. It provides COS interface and utility routines necessary to support the production of portable programs. This includes format conversion routines and machine-dependent constants, for example.

3.10 Important Subsystems

Subsystems are components which use system-provided features to support user services. Two important subsystems in the initial implementation of the Cronus systems are the user interface and the monitoring and control subsystem.

The user interface consists of several components, including the session manager, command interpreter and terminal manager. The user may gain access to the system from dedicated terminal access concentrators, from one of the shared hosts, or over the internet. The interactive processes which are controlled by the user interface will be distributed across the cluster as required either by the application itself or under the direction of the user. A discussion of the user interface may be found in Section 11. In addition, examples of user interaction are shown in Appendix A (Scenarios of Operation).

The monitoring and control subsystem (MCS) makes it possible for an operator to monitor and control the entire cluster configuration from a single console. The functions of the MCS include starting or restarting parts of the Cronus configuration, monitoring its facilities and components, and collecting error reports and statistics. The MCS monitors object managers and collects statistics based on a functional decomposition across the Cronus configuration rather than a site-based decomposition. The monitoring and control design is described in Section 12.

3.11 The Layering of Protocols in Cronus

The underlying support for the Cronus cluster architecture is a high-speed local area network. The Ethernet standard has been selected for an inter-host transport medium within the initial Cronus configuration. The Cronus design does not, however, depend directly on this, so later versions may use a different local network. Furthermore, the design does use the DoD standard protocols at higher levels, and requires an interface between them and the physical local network.

To accomplish these objectives, we have developed a Virtual Local Network based on DoD Internet Protocol (IP) conventions and a representative set of local area network capabilities. The Virtual Local network is an interhost message transport medium which is independent of the physical local network.

The Virtual Local Network layer is described in Appendix C. It provides a primitive datagram service, compatibility with Internet addressing, and independence from the details of the physical local network. VLN datagrams can be specifically

addressed, broadcast, or multicast. The VLN guarantees that datagrams are delivered in order (sequenced) when they are delivered at all, and that a datagram is received once or not at all by each intended recipient (non-duplication).

4 Object Management

4.1 Introduction

In this section, we outline the Cronus object model and show how it is used to structure the kernel of the system. This discussion consists of the following elements:

- A short discussion of the object model in general, and of its relationship to Cronus objects.
- A general description of the basic objects that are included in the first implementations of Cronus.
- The system primitives that Cronus uses to cause operations to take place on objects.
- The role of special processes, called object managers, in the implemention of objects.
- The mechanization of the Cronus primitives, and the role of the operation switch in this mechanization.
- The definition of generic operations that are defined for all Cronus objects.
- The structure of object managers.

In the course of this section, it will be necessary to refer to the characteristics of Cronus processes, and to the methods of communicating between such processes. Those elements of process management and interprocess communication which are needed for the understanding of the Cronus object model and for the construction of object managers will be sketched in this section, while the details have been placed in Sections 5 and 6.

4.2 General Object Model

There is a considerable and growing literature concerning object models and object-oriented programming, and it is not our purpose to describe these methods in detail. On the other hand, the conceptual framework and terminology of object-oriented programming and system decomposition has not fully stablized, and any system, like Cronus, that claims to use this methodology is actually selecting from a range of ideas and applying them to a specific situation; in this case, to the design and implementation of a distributed operating system.

The basic idea of object-oriented systems is that all interactions can, at some level, be described in terms of a set of defined operations on objects. These methods are strongly associated with the development of the Smalltalk-80 system [Goldberg 1983], but are also an outgrowth of work in the manipulation of data abstractions [Liskov 1977], [Robinson 1977], and recent developments in programming languages. There are useful, brief introductions to the use of these methods in [Jones 1978], [Weinreb 1981] and [Rentch 1982].

At first glance, one might consider it enough to think of an object as an instance of a data abstraction. If the internal structure of the data object is suitably hidden from the outside world and the proper operations provided to manipulate the object, we can find out everything we need to know about it and, equally important, nothing about how the object is actually put together. This is a strong application of the hiding principle of software engineering, combined with a set of methods to examine and modify the part of the data object which is of interest to the outside world.

The object model is this and more, however. There are several extensions to this basic idea which have been made in various systems. One of the most important is inheritance, which we will discuss below. Another is the addition of objects which are more than instances of a data abstraction; for example, in Cronus we have process objects as well as pure data objects.

In Cronus, all the objects which are alike in their structure and in the operations which they respond to are members of a Cronus type (in other systems, this is often referred to as a class). Inheritance describes a relationship between types.

We can say that a particular type is a subtype S of some other type T. In saying this, we are saying that an instance of the type S is like an instance of type T in some important way. Usually this is described by noting that any operation which may be invoked on an instance of T may also be invoked on an instance of S. This does not mean that exactly the same procedure will be applied to exactly the same kind of entity. For example, all Cronus objects inherit the properties of the basic Cronus object type CT_Object. There are a set of operations defined on this object, including Remove, which causes the object to go away. A very different procedure is used to Remove a primal file object (whose type is CT_Primal_File) than the one which removes a user process (whose type is CT_Program_Carrier). But there is some clear intuitive feeling which we have of what Remove means if we think of primal files and user processes as objects.

It is worth noting that the inheritance relationship is rather different from the relationship which one finds in composite objects. For example, the Authorization Manager supports the type CT_Group, which is a composite object that is built out of principals (objects of type CT_Principal, which is a representation of a system user) and other objects of type CT_Group. Groups are not subtypes of principals, but are constructed from them. Some operations that can be invoked on a principal, such as the ones which manipulate the group expansion list have no analogue in the definition of a group, and make no sense if they are invoked on a group.

The following are the basic object types that constitute the initial implementation of Cronus:

CT_Object: This is the most basic type, and the generic operations that create and remove objects and maintain the access control lists and object descriptors (see Section 4.4 and Cronus User's Manual object(3)) are defined for objects of this type. In Cronus this is an entirely abstract form, and there are no instances of objects of type CT_Object.

CT_Host: The Cronus system is made up of a series of hosts which provide services for users. This object has a process component that creates and manages the primal processes that, in turn, actually perform the services and manage the other objects of the system. The CT_Host

object is sometimes called the Primal Process Manager for the host, because that is its most visible function. The CT_Host object is closely allied with the operation switch, which is used to implement the invocation of operations on objects.

CT_Primal_File. The initial implementation of Cronus supports files which are bound to a specific host. All ordinary user data is stored in objects of type CT_Primal_File. In addition, a number of other object types are constructed from primal files.

CT_Catalog: The Cronus catalog is made up a series of entries which translate symbolic names into the corresponding UID.

CT_Directory. The Cronus catalog entries are organized into objects of type CT_Directory. These are built from objects of CT_Primal_File, but this structure is entirely hidden from the user by the Catalog Manager.

CT_Principal. A principal is the system's representation of a user or a system service which requires access to some other service or object manager. The access control system depends on identifying the objects of type CT_Principal which are permitted to carry out an activity.

CT_Program_Carrier: A program carrier is a process shell that is prepared to receive a user program. The basic primal process is too simple an object to be effectively used for applications, even though it is adequate for long-lived independent processes like object managers.

There are a number of other object types which are associated with the Catalog Manager (such as CT_Symbolic_Link) and with the Authorization Manager (such as CT_Group), but the system could function without them.

In object-oriented programming, a client invokes operations on an object, often called the receiver, which is identified by a UID, ObjectUID(3). The operation itself may be represented as a

(3). There are a few cases in Cronus where objects are

pair

<OperationName, Parameters>

In Cronus the basic primitive which causes an operation to be invoked on an object is InvokeOnHost. This causes Operation to take place on the object named by ObjectUID on a host at a specified network address. The operation switch of the Cronus kernel provides the mechanization of this primitive (see Section 4.5)

While the primitive InvokeOnHost is sufficient to support the system, the relatively large number of reply messages suggest that there should be a more efficient method for answering a request(4). A second message primitive, SendToProcess is provided for this purpose. When a message from a client is delivered, the ProcessUID for the client is included. The manager may then use SendToProcess to reply directly to the client.

In a distributed system, the client does not usually know which host has the object manager which is responsible for a particular object. Each object must be willing to say whether it is on a particular host; that is, there is a particular operation, called Locate that is among the operations which is defined for every object in Cronus. When this operation is invoked on the object ObjectUID at some HostAddress, the object manager for that type will reply if it manages that object (5).

If the client does not specify the host when invoking the operation, the PSL performs the required Locate operations to determine where to send the operation. These Locate operations

---

identified by other means, for example, a specific catalog entry may be identified by the symbolic name which is being manipulated. The argument presented is analogous, so it is sufficient to consider the cases where the object actually has a UID.

(4). If InvokeOnHost is all that is available, the reply must be passed through the manager of the process to which the reply is directed.

(5). Actually, if the client wants the negative acknowlegement, it will also reply if it doesn't manage the object.

are often performed using the broadcast facilities of the VLN. The PSL (or the client) may cache locations of specific objects and object managers for increased efficiency. In addition, primal objects, which are bound to the host which creates them, can be found quite easily. The PSL looks at the HostAddress portion of the UID, which contains the address of the host which generated the UNO portion of the UID. For the current implementation, the UNO is generated on the host that creates the object, and that also currently holds the object if it still exists.

Subtype relationships are not a primitive concept in the implementation of Cronus. There is no direct implementation of inheritance; there is, instead, a discipline which says that the manager of each subtype must implement the inherited operations. Subtype relationships are statically realized in Cronus, through the cooperation of the object managers and the operation switch. In addition to simple re-implementation of the inherited operations (which is used for the generic operations), there are several static implementation techniques that can achieve inheritance. A manager may register several type values with the operation switch, and implement some as subtypes of the others internally. Alternatively, one manager may invoke another through the standard mechanisms.

4.3 Object Naming

The Cronus object model requires a mechanism for delivering messages addressed to objects. This mechanism, outlined briefly in Section 4.2 and described in detail in Section 4.5, is called the operation switch. The operation switch, in turn, requires the client to identify the object which is being modified or examined. The standard identifier for an object is its UID, which is a bit-string containing 96 bits. This bit string consists of two components: a unique number (UNO) that is different for each object which has ever existed in the cluster, and the Cronus type. It is useful to think of the UID as having four fields (see Cronus User's Manual uid(4), uno(4)):

HostAddress. the 32-bit Internet address of the host which created the object. If the object is a primal object, the HostAddress is also the actual address of the object.

if it still exists.

IncarnationNumber: a field containing an integer which is incremented whenever the host is loaded or reset, or when the associated SequenceNumber field overflows.

SequenceNumber: a simple counter field which is used to assure the uniqueness of each UNO that is used to name an object.

CronusType. the 16-bit integer specifying the Cronus type of the object

Between them, the IncarnationNumber and SequenceNumber fields contain 48 bits, but the subdivision of this string may vary from host to host, for the hosts in in the initial implementation, each field is 24 bits long.

It should be observed that the object is actually identified uniquely by the UNO portion of the UID, and that the the Cronus type is added so the operation switch can find the object manager. In particular, it is possible to think of an object as having more than one UID, consisting of the same UNO paired with different types. The current system does not make any interesting use of this possibility.

There are also generic (or logical) names, which consist of a zero UNO and a type field specifying the type of the generic name. Specific names are used for objects which can be created and destroyed, and have private state information which is important to the accessor (e.g., a particular file). Generic names are used for special purposes. For example, the client can find out if there is an object manager for a particular type on a host by performing an InvokeOnHost to Locate the generic name. Generic names are also used in operations, like Create, in which there is no object name available; the generic names act like class objects in other object oriented systems like Smalltalk, or like the generic addressing facility in NSW's MSG, which is used to address an instance of a service.

The PSL provides a pair of functions which convert between a type name and the generic name for that type (see Cronus User's Manual uidtype(2)). Generic names, like types, can be referred to symbolically. By convention, logical names begin with the

prefix "CL_". For example, CL_Primal_File is the generic name of an object of type CT_Primal_File.

Accessing agents interact with object managers using Cronus Interprocess Communication. The client may initiate access by giving either the UID for the object or by giving its symbolic name. The PSL provides functions which will accept either name. If the accessing process has the UID of the object, the PSL simply constructs a message that invokes an operation upon it. The operation switch delivers the requested operation code, 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. If, on the other hand, the accessing process does not have the UID, the PSL first consults the Cronus catalog; then, when it knows the associated UID, it composes the message and sends it on its way.

This means that we allow the symbolic catalog to be by-passed when an object is accessed, and the accessing process knows the UID. This improves performance and enhances the flexibility of using primitive objects to build complex objects, since the object manager for the complex object can use the UIDs of its components directly. The cost of achieving these benefits is primarily one of increased implementation complexity:

1. Access control is performed in a decentralized fashion by all of the object managers.

2. Information about objects is distributed among object managers and catalog managers. Care must be taken to ensure that the information about an object is consistent, or if it is not, that the system can operate properly.

4.4 Generic Operations On Objects

The generic operations are defined for all system objects. These operations fall into several groups:

Create and Remove. These bring the object into existence and destroy it. The operation Create is invoked on the

generic name for the object. These operations must be defined for all objects.

Locate: If the object exists and is managed by the object manager which receives the message, the manager replies that it knows about the object. This operation must be defined for all objects.

Read_ACL and Write_ACL: These manipulate the access control list of the object. These operations must be defined for all objects which are separately access controlled. There are a few objects whose access is controlled through another object. For example, objects of type CT_Catalog_Entry are controlled through the permissions on the containing object of type CT_Directory.

Read_Sys_Parms, Write_Sys_Parms, Read_User_Parms, Write_User_Parms. Every object has an associated object descriptor. The object descriptor contains various pieces of information about the object that are made visible to the outside through these Read operations, and may be modified by the Write operations. Access is controlled separately for the User and Sys portions of the object descriptor.

Report_Status: This operation is normally performed on a generic name associated with an object type. For example, Report_Status is invoked on the generic name CL_Primal_File to find out how much space there is available on the associated file system.

For some operations, such as Create, the exact list of parameters and responses will vary from object type to object type. Other operations, such as those which operate on the access control list, perform in the same way for all object types. For details, see the appropriate sections of the Cronus User's manual, especially object(3), acl(3), the descriptions of the objects below and in Section 3 of the Cronus User's manual, and the descriptions of the PSL routines in Section 2 of the Cronus User's Manual.

4.5 Object System Implementation

In order to describe the design of the operation switch and its role in message-oriented interprocess communication, we must briefly introduce Cronus processes (the Cronus process is described in detail in Section 5).

Cronus processes are constructed from constituent host processes (CHPs). The properties of a CHP are defined by the machine architecture and the constituent host operating system (COS). The Cronus process is constructed from one or more CHPs, with the addition of Cronus process features. The simplest type of Cronus process is the primal process (PP). A primal process is a CHP which can invoke operations on objects through the Cronus Interprocess Communication facility and can be controlled by the Primal Process Manager. In addition, a primal process can use the Cronus primitive Receive to receive messages sent through the Cronus IPC by either InvokeOnHost or SendToProcess.

The implementation of Receive employs CHP-specific synchronization facilities, described in the appendixes on the interface to the COS, to build an asynchronous Receive operation.

This section describes the framework of the object system implementation on Cronus hosts. Figure 4.1 illustrates the relevant components on a single host. The boxes in the figure represent abstract modules of the implementation, and do not necessarily map one-to-one into CHPs or address spaces.

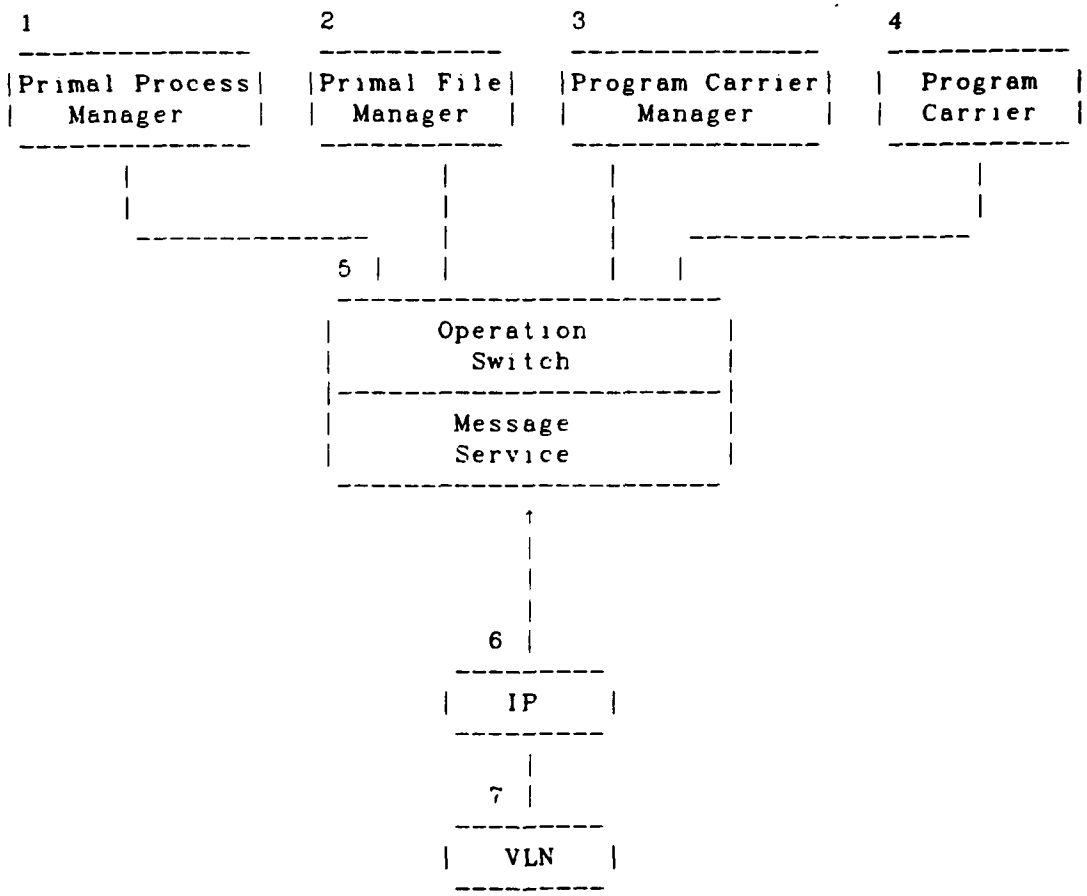


Figure 4.1 Object System Components

In Figure 4.1, boxes 1-4 are Cronus process objects; box 5 is the operation switch, which accepts messages from and delivers messages to the Cronus processes on this host, box 6 is the IP protocol demultiplexing service, and box 7 is the Virtual Local Network layer.

The operation switch is table-driven. This table contains routing information that the operation switch uses to direct messages from process to process. The sender and receiver may both be on a single host, or the message service may be involved in a host-to-host message transfer. The operation switch does not retain information about the messages, although it may gather statistics and transmit them to the Monitoring and Control System (see Section 12).

Since the invoker can request reliable message transport, and ordinarily does so for InvokeOnHost applied to a specific host address, a failure of an operation invocation is not likely to be due to a transient communication fault, with high probability, either the network or the target host, or both, are down (see Section 6 for a detailed description of the IPC and these services).

The invocation sequence for an operation is:

- The Cronus Process Support Library (PSL), which is the component of the system that appears within the client process, formats a message which contains the name of the object, the operation, its parameters, and other information which is needed by the system.

- The message, which is marked as an invocation of the operation, is handed to the local host's operation switch. If HostAddress specifies the local host, it processes the message itself; otherwise, it forwards the message to the specified host. (These functions are directly supported by the Cronus Interprocess Communication facility, which is described in detail in Section 6.)

- The receiving operation switch examines the ObjectUID, determines the type of the object, and hands it to the object manager for that type, if there is one.

- The object manager for the object type then performs the processing associated with the operation and its parameters.

- Although it is not necessary for an operation to follow a request-reply paradigm, most do. If a reply is needed,

the object manager prepares a message that is returned using the SendToProcess primitive.

Figure 2 illustrates the transmission of an operation from the invoking process, through the local operation switch, to the remote operation switch, and finally to the receiving process. This section describes the calls and the representation of data structures at the interfaces 1, 2, and 3.

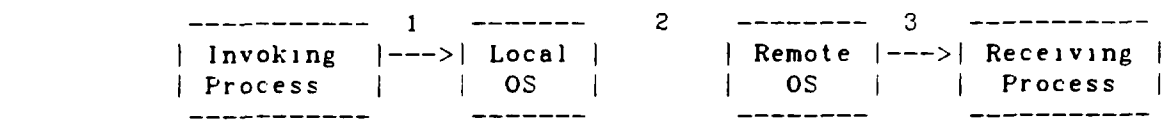


Figure 4.2 Operation Switch Interfaces

When the client performs an InvokeOnHost primitive on the Cronus object, a message is generated that is ultimately directed to a manager process and accepted by a Receive in that process. Information crosses interfaces (1) and (3) by means of Cronus system calls, which are representations of the primitive functions, made by the invoking and receiving processes, these calls may be represented as:

```
InvokeOnHost(TargetAddress,ObjectUID,Operation)

Receive(SourceAddress,SenderUID,ObjectUID,Operation)
```

where the function parameter Operation includes both the intended operation and its parameters. (6).

(6). The calling sequences for these functions have been modified for purposes of presentation clarity, see the Cronus User's Manual send(2) and receive(2) for a description of the actual calling sequence.

Interface (2) is peer-to-peer communication between operation switches, which is discussed in greater detail in Section 6. Messages exchanged between operation switches are octet sequences. The Operation parameter of the InvokeOnHost call is not interpreted by the operation switch, and is treated simply as data to be moved. The message has several header fields that are visible to both operation switches; these include the UID of the object being operated upon (ObjectUID) and of the client (ProcessUID).

When the InvokeOnHost message arrives at the target host, the operation switch tries to map the type to a manager process on the host. The table of possible destinations consists of a list of generic UIDs for ordinary managers and specific UIDs for objects which are managed separately (7). The operation switch first checks the ObjectUID against the list of specific UIDs, then the Type field against the list of generic UIDs. If the mapping is not successful, the invocation is discarded, but will generate an exception reply. If the mapping is successful, the message is transmitted to the manager process. The manager obtains the information by initiating an ordinary Receive request, when the Receive completes, the SourceAddress, InvokerUID, ObjectUID and Operation have been made available to the manager process.

Although one can reply by invoking the Send operation on the object ProcessUID, replies are usually sent by means of the alternative SendToProcess primitive. This primitive hands messages addressed to a specific process across interface (1). The operation switch then marks the message which it ships across interface (2) as a SendToProcess message. The receiving operation switch then places the message on the queue for the target process, bypassing its object manager. The mechanism for delivery, Receive, is independent of the transmission mode of the original message.

---

(7). Currently, the only example of such a separately managed object is the virtual terminal in the user interface (see Section 11).

4.6 Object Manager Structure

Object managers are asynchronous independent processes. They are asynchronous because they interleave the processing of messages. An object manager often invokes operations on other objects to satisfy the requests it receives; it does not wait for the reply to such a request, but moves on to the next request or reply from a previous operation. They are independent processes because they are daemon processes which are started by the system (or its monitoring and control section) or by another daemon process. They receive messages, originate requests to satisfy the client requests, and reply to the original messages.

The asynchronous character of the object manager has a significant impact on its structure. Managers receive messages which cause them to undertake actions. These actions may be of two types. The first type occurs entirely within the manager's own address space (or within a single Cronus process that may consist of more than one COS process), and is called a local action. The second type requires the manager to perform one or more operations, called *secondary* requests, on objects that it does not manage. It must be able to keep track of a number of these actions. On the other hand, the manager cannot wait for the response from a secondary request before it accepts its own next request. The processing that comprises the operation is divided into portions that are performed before and after the secondary request is issued. When the manager issues the secondary request, it saves components of its state that are needed to complete the processing when the reply arrives.

There are a number of common elements in the construction of object managers:

> A manager normally consists of an initialization section and a main loop which is driven by the arrival of requests through the Cronus interprocess communication facility. Since a manager normally runs forever (until the system crashes), there may not be code for wrap-up.
>
> The manager parses incoming messages, and dispatches on the message class, which takes on the values Request, Reply, and InProgress.
>
> A new Request message causes the manager to set up a control

block for the operation.

A Reply message causes the manager to identify the control block associated with the message, and to continue processing as required by that message.

In the case of a local action, the manager receiving the message will (normally) process the request to completion and compose a reply to the originating process.

If a secondary request is necessary, the situation is similar to that found at the originator. A request can be put into the form.

```
InitialPortion
Op(Obj) -> Reply
PostProcessing
```

That is, a secondary request is basically some operation (Op) on an object (Obj) which generates a Reply. Before we invoke this operation, we usually have some initialization beyond composing the message (InitialPortion) and after we get the reply, we often need to do some PostProcessing.

The procedure that invokes the operation also creates a control block that contains the information required for reply processing. After it passes the invocation to the IPC mechanism, it returns without waiting. The manager then processes the next IPC message (which may be a Reply from a secondary request, or a new Request), if there is one available. Otherwise, it goes to sleep until the next message arrives (see Section 6 and ipcmisc(2) in the Cronus User's manual for details). When a Reply for a secondary request arrives, the manager finds the control block associated with it, and performs the reply function. When the reply processing returns normally, the PostProcessing routine is invoked if the message is marked OK, and an alternate error-handling routine is invoked if the message is marked NOT_OK.

The independent character of the object manager principally effects the way errors are handled. When a process is interactive, it makes some sense to report the error to the user. If an independent process detects an error condition, it may be

necessary to report the error to the client that issued the request, to the monitoring and control station (MCS, see Section 12), or to both. In addition, Cronus managers keep statistics on the kinds of errors which have been detected, and report them to the MCS periodically.

A manager that encounters a failure during an operation, particularly when there are secondary operations involved, must take steps to assure that the information which is retained across host crashes (the permanent state of the system) and any internal status information (the temporary state of the system) are correct and consistent.

Changes in the permanent state of the system are made by atomic transactions. If it is necessary to make several changes in the recorded data to perform an operation, the manager that receives the operation assures the client all the changes will take place or none of them will. That is, in the case of a failure, the atomic transaction mechanism either forces the transaction to completion by carrying out the intentions which have been posted, or undoes those portions of the intentions list (see Cronus User's Manual intent(2)) already marked as performed.

When a manager (or any other process, for that matter) is carrying out a composite action consisting of more than one operation on one or more objects, there are often other changes in temporary state which must be undone if an error is detected. The process maintains a work-in-process list that contains an entry for each action that is not yet complete. For example, if a process has acquired locks on several files, and discovers that an additional lock which is needed cannot be acquired, the original set must be released. The work-in-process list also contains entries for additional special processing that is required if the action does not complete (see Cronus User's Manual wip()).

5 Process Management

5.1 Introduction

Processes are the active portion of any system. Each host and constituent operating system in a Cronus cluster has at least one natural concept of the process. More generally, several different kinds of processes are present in each host, fulfilling different roles. In the absence of a distributed operating system, the processes on two hosts are unrelated to each other. This section describes how Cronus processes work and how they communicate with each other. The details of how processes are constructed from constituent host processes (CHPs) are discussed in Appendixes D, E, and F. In the following discussion, it is usually safe to visualize a Cronus process as being built from a single CHP with the addition of an object descriptor and some specialized facilities which make Cronus work. On the other hand, the implementation might be quite different in reality. That is, a Cronus process might be made up of several CHPs, or a CHP might include more than one Cronus process (8).

If we wish to build a system of cooperating processes on a cluster of computers, and to use it as a base for a distributed operating system, we must do the following:

- Define a standard method for communicating among the processes. Cronus treats processes as objects, and uses the standard Cronus IPC facility and the primitives InvokeOnHost and SendToProcess for all interprocess communication. All procedures developed for structuring and parsing messages for operations on objects, such as those described in Section 6, may be used for manipulating process objects as well.

- Establish mechanisms for creating and controlling processes on hosts of different sorts. Again, since Cronus processes are objects, this reduces to the definition of the operations which may validly be applied

(8). In fact, a Cronus process might even span hosts. In the current system design, all Cronus process are primal processes, that is, they are bound to a single host. Later implementations may relax this restriction.

to the process objects.

- Provide a method for organizing the process objects to perform tasks. This is accomplished by defining other objects which reflect the required organization. The collection of processes on a host, for example, is represented by an object of type CT_Host, which will be described below. Another example are those processes that make up a user session, which are represented by an object of type CT_Session_Data (see Section 11).

The following three Cronus types are discussed in this section:

- CT_Host: the organizing object for the primal processes associated with a physical host.

- CT_Primal_Process: the most fundamental type of process. Object managers are normally constructed from processes of this type.

- CT_Program_Carrier: a subtype of CT_Primal_Process that has augmented process control facilities that make it more suitable for implementing user processes.

There is one object of type CT_Host associated with each physical host, and it is the object manager of the processes of type CT_Primal_Process on that host. It is responsible for starting up Cronus services, which are also object managers for the basic system objects; it is also responsible for gathering the information which the operation switch needs to route messages to the other object managers and to specific processes when the primitive SendToProcess is used.

There are two basic Cronus process types, CT_Primal_Process and CT_Program_Carrier(9). The type CT_Program_Carrier is a subtype of CT_Primal_Process. Ordinary primal processes lack essential process control functions and other desirable characteristics needed for application programming. The subtype

(9). Future system versions will introduce additional process types which may be distributed in extent and have special reliability properties.

CT_Program_Carrier provides an environment tailored to the requirements of application programs.

Primal processes and program carriers never migrate; once created, the process remains on the same host until it is destroyed. The HostAddress in a UID for a primal process or program carrier tells where the process is, so an operation switch can tell exactly where to deliver a message addressed to it.

Every host participating in the system must support an object of type CT_Host, which is also referred to as a Primal Process Manager (PPM), and primal processes. In their minimal forms, the host object and primal processes are relatively simple. This keeps the cost of integrating a host type into a Cronus cluster low for those minimally integrated hosts that can obtain system services from other hosts, but do not provide system services.

A primal process which plays a well-defined functional role within the system is called a Cronus service. Cronus services are object managers for system-defined object types, for example, a Primal File Manager or Program Carrier Manager.

Cronus processes may make use of some or all of the functions in the Process Support Library (PSL), which provides high level interfaces to many system functions as well as general purpose utilities for interfacing to and manipulating the Cronus environment. Portability is a major goal for the PSL, so that it can be implemented readily in whole or in part on new host types. The PSL is discussed further in Section 5.4.

5.2 Objects of type CT_Host

The basic organizational elements of Cronus are objects of type CT_Host. These objects correspond to the intuitive physical hosts that make up the Cronus cluster. A CT_Host object consists of the the Primal Process Manager for the host and the basic tables which are used by the operation switch in routing operation invocations. In some sense, it is reasonable to think of the operation switch itself as a part of CT_Host. When a host joins the Cronus network, only the lowest level of network

software is functioning; the Monitoring and Control System (See Section 12) engages in a dialog with this primitive host element, and brings up the object CT_Host. The MCS is therefore the object manager for the objects of type CT_Host.

The Primal Process Manager (PPM) component of a CT_Host object implements operations concerning primal processes as a class. The tables that identify the object managers and processes that are on a particular host, and that therefore are used to implement the Cronus primitives InvokeOnHost and SendToProcess, are maintained by the Register and Delete operations on the CT_Host object.

In addition to the generic operations (see Cronus User's Manual object(3)), the following operations are defined on objects of type CT_Host (see Cronus User's Manual cr_host(3)).

Cronus_Restart
Service_List
Process_List
Register
Delete

The Cronus_Restart operation is used to shutdown all activity on the CT_Host object. It removes all active processes, including the process implementing the CT_Host object itself. After a Cronus_Restart, the host is in a state from which it may be bootstrapped.

The Service_List operation is used to find out what kinds of service the host is prepared to support, and which ones are in fact being supported. The names of these services, which are called role designators, are used to start primal processes that perform the service (see Section 5.3).

The Process_List operation tells what processes are active and what roles they are playing; this is the information which the operation switch has about processes active on this host. Whenever a process is created or removed, the tables must be updated. These tables contain the following entries:

- generic names for objects paired with the specific UID of the Cronus process;

- specific UIDs for process objects that will receive messages through SendToProcess, and

- specific UIDs for those objects whose manager cannot be identified by reference to a generic name (see Section 11).

The tables also contain any COS specific information needed to communicate with the process. They are automatically updated for processes which are created by the CT_Host object itself, such as the object managers. Other processes are created by other managers, for example, the program carrier manager. These inform the CT_Host of changes through the Register and Delete operations

5.3 The Operations on Objects of Type CT_Primal_Process

Objects of type CT_Primal_Process are among the most basic in Cronus. The three system primitives (InvokeOnHost, SendToProcess, and Receive) are defined for these objects. In addition, the generic operations (see Section 4.4 and Cronus User's Manual object(3)) are defined. The particular characteristics of these operations, when invoked on primal process objects, are described in detail in the Cronus manual (see Cronus User's Manual p_process(3)).

The Create operation takes a role designator as an argument, and starts a new primal process performing this role. The role designator may be in one of the following forms:

1. A Cronus generic UID name for the service.

2. A Cronus symbolic service name. These are character strings containing the literal characters of a logical name, for example "CL_Primal_File".

3. A host dependent role designator. These are arbitrary strings, which have meaning only to the PPM on a specific host.

Role designators of kinds (1) and (2) are paired, and are registered with the Cronus system administrator as the names of

standard Cronus functional units. The allowable list of role designators of these kinds for a particular host object may be obtained by invoking the operation Service_List on the object. These primal processes are automatically registered, which makes the logical name known to the operation switch on the host, so that the process can be generically addressed.

Designators of kind (3) provide for the activation of host-specific programs or devices. The host dependent role designator might be a COS-dependent file that is executed as a result of the Create operation. Primal processes created with a host-dependent role designator generally have no associated logical name, and cannot be generically addressed.

The primal process will initialize its state entirely from non-volatile storage (local or remote disks).

A process may invoke any operations on itself as the target object. A process may send itself messages, remove itself, or read or change its descriptor in the same way it performs these operations on other objects.

The operations defined on primal processes provide process control functions. For example, Remove is invoked to "destroy" or "kill" the process. It erases all record of the process state from the system and frees any resources dedicated to the process.

A process which is removed is not notified of the operation, and has no opportunity to terminate cleanly. Only the resources actually used to implement the process object are freed; resources held as a result of the computational activity of the process (e.g., locks on remote files) are not freed. Some primal processes may possess dedicated resources, and Remove disables the process, without releasing these resources.

A reply will be generated to the invoker to indicate that the process has been removed. After receiving the reply, the invoker knows that operations using the UID of the process will not succeed.

The process descriptor is the object descriptor portion of the Cronus process. It is useful to think of the process descriptor as a list of (key, value) pairs, in the sense of the MSL (See Section 6.2 and the list of standard key names in the

Cronus User's Manual keys(4)). Some of the values implement process control. For example, the pair (Key_Priority,5) would indicate the importance of a process relative to other processes for competing resources. Some keys must be present in the list ("required keys"), while others are optional (see Cronus User's Manual p_process(3), process(4)).

All process objects must respond to the required keys in a uniform way. If an object supports a standard optional key, the process must apply it in a uniform, system-wide manner. Additional, elective keys may be present. Their interpretation is not specified by Cronus, but is the responsibility of the process and the other processes with which it interacts.

Currently, the required keys for Primal Processes are Key_MyUID, Key_MyAGS, and Key_IPCEnabled.

The value associated with Key_MyUID is placed in the descriptor when the process is created, and is never changed thereafter. It is the specific UID of the process, and has type CT_Primal_Process (or CT_Program_Carrier, in the case of program carrier objects).

The value of Key_MyAGS is the access group set, used with access control lists to determine access rights to objects at operation invocation time. The initialization and use of access control and authentication data is discussed in detail in section 7.

The value of Key_IPCEnabled controls communication through the operation switch. If the value is true, the process can send and receive messages in the normal fashion. If it is false, the process may not send or receive messages, or invoke operations on Cronus objects. This feature can be used for managing access to network resources.

Currently, the only optional key defined for a Primal Process is Key_Priority, but others may be defined later.

The generic operations on object descriptors permit a process to inspect or modify the descriptor of another process. If several processes invoke these operations on another process at the same time, the effect will be as if the operations were processed sequentially, i.e., they are atomic with respect to

each other.

Since the CT_Host object is implemented by a Primal Process, these process control operations apply to it. One of the operations, Remove, has a special meaning when applied to the CT_Host. Because it is the manager of Primal Processes, removing the CT_Host removes all Cronus processes on the host. This forces a shutdown of the Cronus system on the host.

5.4 Program Carrier

The type CT_Program_Carrier, which is designed to support user programs, is a subtype of CT_Primal_Process, and all of the characteristics of primal processes are inherited by program carriers. Additional operations can be invoked on program carrier objects, and the set of required keys in the process descriptor is enlarged. The program carrier

- provides a process which can be created, loaded with a program, started, and stopped under remote control;
- provides uniform monitoring and debugging support; and
- provides application developers with the ability to control a collection of user written (possibly distributed) processes.

A Cronus host is not required to support the CT_Program_Carrier process type; however, hosts which are not dedicated to system service roles usually support program carriers.

The generic operations (see Cronus User's Manual object(3)) are all defined on objects of type CT_Program_Carrier. In addition, the special operation Search_All_Descriptors is defined on the generic program carrier object.

Create creates a new process of type CT_Program_Carrier and returns the UID to the invoker. The program carrier manager initializes the process descriptor of the new process. Several of the fields have default values, in particular the standard input, output, and error output, and the access rights will be

inherited from the invoker if they are set for that process.

Once a process has been created, the parent (or another process) may alter values in its process descriptor, using the generic operations on the object descriptor, if it has the appropriate permissions.

The Report_Status operation may be invoked on the generic name CL_Program_Carrier to test for the availability of resources before performing the Create operation. Resources may include processor type, primary memory size, and special processor capabilities, such as floating point hardware. This operation is used as part of the scenario for selecting a site at which to run a program (see Appendix A.8).

The Search_All_Descriptors operation may be invoked on the generic name CL_Program_Carrier to find all program carrier processes on a host with the designated key-value pairs in their descriptors. Two important uses of this operation are: 1) a search on the Key_Session key-value pair, to locate all process associated with a user session, 2) a search on the Key_Thread key-value pair, to locate all processes belonging to a thread.

Cronus supports several kinds of relationships among program carrier processes. All processes belonging to a session are related, and can be located as a group; processes are related in parent-child relationships; and processes are bound together by the data streams that connect standard input and standard output (and by other streams that may be explicitly opened by the processes).

The knowledge that a group of processes belong to the same session is useful for coarse-grained error recovery (killing the session). Streams are used primarily to provide continuous data paths between processes.

The parent-child relationship supports the flow of control information among processes. When a program carrier is created at the request of another program carrier, the list of children in the requesting process's descriptor is updated, and the requesting process's UID is entered as the parent in the new process's descriptor. When a process is removed, a message is sent to its parent. The parent can then use that information to notify or terminate other children that were communicating with

the first process. As a result, the processes form a tree; any subtree of this is called a process group, and the program carrier manager supports operations on process groups as well as on processes. these operations are applied to each process in the subtree named by the process that the operation is invoked upon. These operations reduce synchronization requirements at process start-up, and still provide an easy mechanism to control all the children of a process.

The operations defined on objects of type CT_Program_Carrier are described in the Cronus Manual (see Cronus User's Manual prog_carr(3)). in addition, the operations on its supertype, CT_Primal_Process (see Cronus User's Manual p_process(3)) and the generic operations (see Cronus User's Manual object(3)) can be invoked on program carrier objects. The operations that are specific to the program carrier objects are:

Clear_Program
Load_Program
Proceed
Suspend
Stop
Report_State
Change_State
Breakpoint
StopGroup
SuspendGroup
ProceedGroup

These operations are sufficient to meet two basic objectives: 1) It is possible to load a binary image into a new program carrier object, start it, and allow the process to complete or be cleanly stopped; and 2) the Suspend, Proceed, Report_State, Change_State, and Breakpoint operations, together with the Primal Process operations, will support general remote process control.

The required keys for the object descriptor of a program carrier are described in the Cronus User's Manual, on prog_carr(3) and process(4). These include:

- Key_MyUID, Key_MyAGS, Key_IPCEnabled, and Key_Priority, all of which have the same meaning for program carriers

as for primal processes.

- Key_State, which informs other processes of the current state or mode of a process. The states reflect only the interactions of Cronus operations and the process object, and do not capture finer state subdivisions which are host or local operating system dependent.

- Key_StInput, Key_StOutput, and Key_StErr identify the data streams that are used for standard input, output and error reporting. The streams are used in a manner analogous to the standard input and standard output of the UNIX process model. See prog_carr(3) for a detailed discussion of the mechanism for input/output redirection.

- Key_Parent, which is the UID of the process which requested the creation of this process.

- Key_Children, which are the processes, if any, created directly at the request of this process.

- Key_Thread, which is a UID identifying the portion of the user session in which this process was created. A user session may consist of one or more threads of activities that may be running in parallel.

- Key_Terminal, which is the UID of the virtual terminal, if any, that is associated with this process.

Since the program carrier object is designed primarily to support user processes, many of the details of the use of these keys are described in Section 11.

5.5 Process Support Library

The Process Support Library (PSL) is a basic part of the Cronus implememention. It contains a large number of functions which can be used to construct Cronus object managers and user programs. All Cronus programs are expected to use the PSL to perform the functions which it supports. The distribution of

responsibilities between the PSL and the Cronus kernel is often not defined, and may shift from implementation to implementation. Any program that bypasses the standard PSL interface, and makes use of private information about this division is no longer insulated from modifications of the definitions of the objects, object managers and the kernel, and the use of such a program may produce unexpected results in the future.

The following is a partial list of the kinds of functions which one may find in the PSL:

- A set of standard interface routines for all operations on the basic Cronus objects. There are two sets of interface routines: those which are designed for use with managers and other asynchronous programs, and which do not wait for the response from an operation; and those which are intended for use in interactive programs, which do wait for a reply if one is expected.

- Functions supporting composite activities, such as writing data on a file specified by a symbolic name.

- Functions supporting the construction of Cronus object managers. These include routines for manipulating UIDs and UID tables, for managing the processing requests and their responses in asynchronous processes, for creating and modifying work-in-process and intentions lists.

- A standard error reporting facility for both asynchronous and interactive processes.

- Sublibraries for message composition, string manipulation, portable input/output operations, and device management.

The PSL is described in detail in Section 2 of the Cronus User's Manual.

6 Interprocess Communcation and Messages

6.1 Overview

Cronus presents 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:

- 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. This facility, which was introduced in Section 4, is further described in the current section.

- Conventions for passing data using Cronus canonical data types permit messages to be composed without concern for the heterogeneity within a cluster.

- 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 InvokeOnHost, SendToProcess, and Receive. These primitives support the asynchronous communication of uninterpreted data octets among Cronus processes, by means of the abstractions of sending to a process or invoking an operation on an object.

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 Object-Operation Protocol, for the patterns of communication that arise in performing operations on Cronus objects.

The IPC and message structure facilities, and their relationship, will be discussed in the following sections. The details of the interfaces and the specific implemenation of the IPC will be found in the Appendixes on the COS implementation and in the Cronus User's Manual.

6.2 Messages in the IPC

The IPC facility supports two classes of messages: reliable messages and minimal effort messages.

> A message sent reliably will be delivered to the receive queue of the addressed process (or the manager of the addressed object on an InvokeOnHost) despite transient failures in the communication substrate. A reliable message will be delivered at most once.
>
> Minimal effort messages are transmitted with whatever reliability characteristics are provided by the communications substrate. The IPC facility does not attempt to provide a sending process with information regarding the disposition of the message.

In both cases, the message is protected by an end-to-end checksum, so if the message is delivered, the content may be presumed to be correct.

The sending process may use minimum effort messages whenever it seems appropriate. The current implementation uses them for all messages sent to a broadcast or multicast address.

Messages may also be categorized by length. A small message will fit into an IPC packet throughout the cluster. The maximum size of a small message is implementation dependent, and in the current system is about 1500 bytes (see Cronus User's Manual message(4)). A large message may have a length set at the time the message is initiated, or the length may be indefinite. Minimal effort messages are constrained to be small, while reliable messages may be small or large.

A large message may be of any size, although they are generally larger than the small message limit, and the PSL automatically selects a small message for messages below the limit and a large message for a message above the limit.

Messages of indeterminate length support Cronus streams, which are uni-directional data channels between a source object (sender of the message) and sink object (receiver). Cronus streams are used to interconnect processes with devices and with other processes. Although data flow on the stream is unidirectional, the implementation of a stream involves transmissions in both directions: from source to sink containing data, and from the sink to source containing flow control and synchronization information.

One objective for the IPC facility is to make the distinction between small and large messages be as small as possible. In particular, the content and structure of the information contained in a message, and any information about a message that is delivered to a recipient (e.g., size, source, etc.) is independent of its transmission characteristics. The sender of a message indicates whether or not the message is to be transmitted reliably, and its length, if it is of bounded length. The receiver need not be concerned with these characteristics of the message.

6.3 Programming Interface

The programming interface for the IPC provides facilities needed to invoke operations on objects, send messages to processes, and receive messages from clients. Many application programs will be written in terms of higher level routines which may be found in the PSL. The interface described in this section

is primarily of interest to systems programmers who are developing and maintaining object managers and PSL routines.

The interface provides direct support for the Cronus primitives (InvokeOnHost, SendToProcess, and Receive), for the full range of message types (reliable small, minimum effort small, and reliable large), and for various buffering strategies that the sending or receiving process might wish to adopt.

When a process invokes an operation on a Cronus object, it uses the PSL function Invoke; when the message is tranferred by the SendToProcess primitive, the process uses the PSL function Send. In either case, the process indicates the size of the message being sent, whether it is to be sent using reliable transmission, and points to a buffer which contains the information which is currently available for transmission. The buffer may contain the entire message or any portion thereof (see Cronus User's Manual send(2)). The IPC accepts the information for transmission, and returns a small integer, called the message handle. If there is more information to be sent, a new buffer is given to the SendMore function, along with the message handle. Finally, the message is completed by applying the LastSent function to the message handle.

The operation switch on each Cronus host provides buffering for messages and synchronization between Cronus processes. Buffering and synchronization are closely related, because buffering in an intermediary influences the synchronization points between processes.

The sending functions accept the message if it can be queued somewhere within the IPC mechanism. It can be in a host-dependent transport mechanism between the process and the operation switch (see Figure 1), on the "receive queue" of a Cronus process (if it is an intrahost message), or on the "network queue" of messages waiting to be transmitted (if it is an interhost message). If the message cannot be queued immediately, it is refused by the IPC, and the sender is responsible for any required recovery.

Even if the message is accepted, the IPC does not report that the message has been delivered or that delivery can be assured. The only way the sender can be assured that a message has been received by it is to wait for a reply from the intended

recipient. Cronus managers respond with at least a ReplyCode whenever an operation is invoked on an object. User processes should normally observe a similar protocol, since lower level protocols cannot assure delivery of messages.

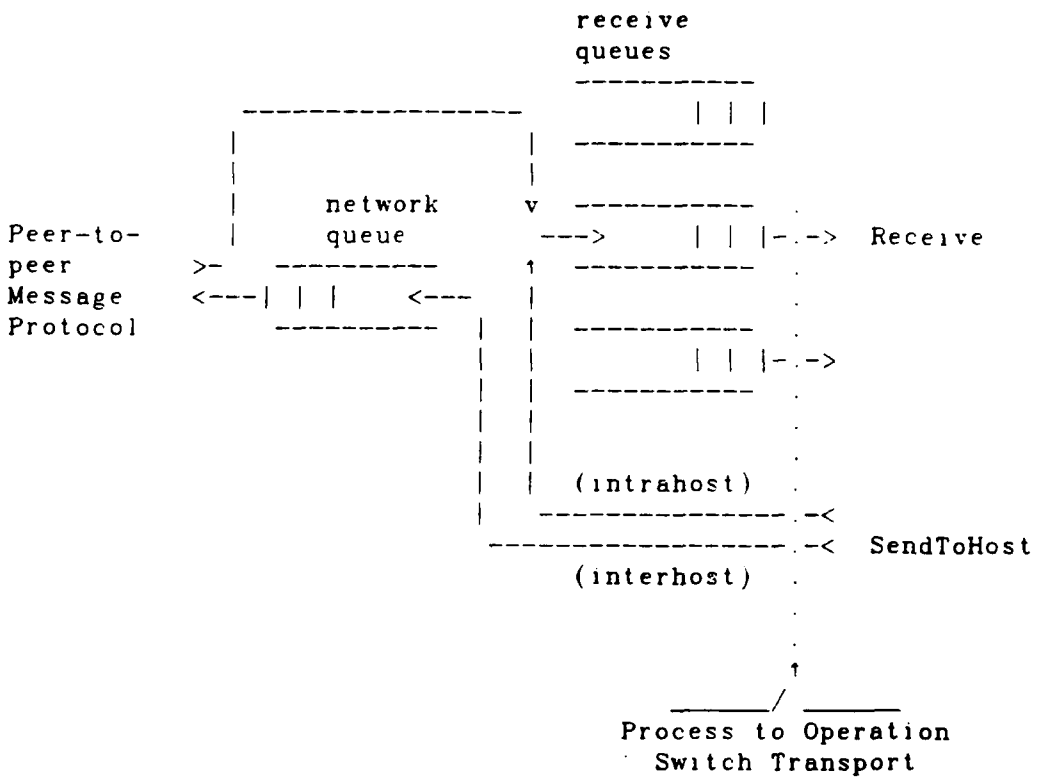


Figure 6.1 Schematic of the Operation Switch

The receive queues are maintained in FIFO order; the network queue is a group of FIFO queues, one per destination host or process. Entries on the receive queues are delivered to client processes to satisfy Receive requests, and entries on the network queue are transmitted to remote operation switches, where they are placed on the proper receive queues.

When the receiving process is prepared to process new data, it executes the Receive or ReceiveMore function. Each new message is started with Receive, and if the entire message is not available, or cannot fit into the buffer that has been given to Receive, more of the data can be read with ReceiveMore (see Cronus User's Manual receive(2)). Both functions return immediately with the data, if any, that is available.

The buffering strategies in the two communicating processes may be different. The sending process can, for example, send the entire message in one piece, and the receiving process may choose to receive it a chunk at a time.

The IPC also provides functions which give the client control over the message queues, the basic timeouts which control error handling, and the processing of asynchronous events (see Cronus User's Manual ipcmisc(2), receive(2)). These functions include.

- WaitForChange suspends the process until an interesting event occurs. Typically, this will be the arrival of another message or more data for a message which has been partially received. Other interesting events include timeouts and events which are unrelated to the IPC mechanism.

- AbortMessage deletes a message from the queue without completing processing (either send or receive).

- SetDefaultTimeout adjusts the standard timeout for the process.

- MsgQueueSize tells how many messages are waiting for processing, including any partially received messages.

6.4 IPC Implementation

The implementation of the Cronus IPC can be described at two levels. There are some elements of it which are generic; the structure of the implementation must support those facilities which clients expect of it. These include the overall issues of buffering, synchronization, and reliability, for example. At the second level, there are specific decisions about how the initial implementation will be constructed. Future implementations of Cronus may choose to do things in a very different way. For example, the current implementation uses the DoD standard connection protocol, TCP, to implement reliable message transport. Future implementations may use a different reliable transport mechanism.

Cronus IPC supports three types of messages.

- small, minimum effort messages;
- small, reliable messages, and
- large, reliable messages.

Neither the protocols used nor the structural requirements of the implementation specify the division of responsibility between the operation switch and the PSL for these various classes of message. In fact, the division might be made differently in different hosts in the same cluster. The transport mechanisms used in the current implementation are shown in Table 6.1.

Small, minimal effort messages are sent from Source Operation Switch to Destination Operation Switch by means of IP datagrams using the standard User Datagram Protocol (UDP). Receipt of an IP/UDP datagram by the Destination Operation Switch is not acknowledged.

On receipt of a datagram, the Destination Operation Switch determines if the enclosed message should go to a local object or process. If so, it places the message on the receive queue of the object manager or process.

TYPE OF MESSAGE	TRANSPORT MECHANISM
Small, minimal effort	IP - Operation Switch <-> Operation Switch
Small, reliable	TCP - Operation Switch <-> Operation Switch
Large, reliable	TCP - One connection per large message, connection establishment initiated by an Operation Switch to Operation Switch interaction, but connection may be in the Operation Switch or the PSL, at the discretion of the host implementation.

Table 6.1 Message Transport Summary

The initial implementation of Cronus will transmit small, reliable messages from Source Operation Switch to Destination Operation Switch over a TCP connection because it is the fastest way to get the implementation working. TCP provides services not required for small reliable messages (e.g., strong sequencing, reassembly) and we may find that the overhead they impose makes the performance of the IPC unacceptable. If this is the case, we will develop a reliable small message protocol (RSMP). RSMP would perform the following services

- Provide receipt acknowledgement.
- Provide for retransmission.
- Perform duplicate detection and elimination.

As with small minimal effort messages, upon receipt of a message the Destination Operation Switch will determine which local object manager or process should receive the message and will place the message on its receive queue.

Large messages are implemented through a TCP connection for each message. There is an interaction between the source and destination hosts to establish the TCP connection. When the message has been transferred, the TCP connection is closed.

The following steps are used to establish a new TCP connection to carry a large message between two processes:

The source host selects the port to be used for the TCP connection, and puts its end of the connection into the listening state.

The Source Operation Switch sends a StartLargeMessage (see Cronus User's Manual message(4)) message over the Operation Switch to Operation Switch TCP connection. This message specifies the destination, the port for the TCP connection, and perhaps the first part of the message.

The Destination Operation Switch places the message on the receive queue of the object manager or process.

When the destination process executes a Receive and finds the first part of a large message, any data sent along with it is delivered. The destination host selects a port for its end of the TCP connection, and uses the TCP port supplied within the StartLargeMessage message.

After the connection is established, the source host will use it to pass message data to the destination host.

After the source process sends the last chunk of data in the large message, the TCP connection will be closed.

This discussion does not specify whether the Operation Switches or the client processes are responsible for managing the connection that carries the bulk of the message data, nor whether the Operation Switches or client processes are responsible for actually using the TCP connection to send and receive message data. These implementation decisions may be made differently for each host type.

6.5 Object Operation Protocol

The Object Operation Protocol (OOP) is used by the PSL whenever operations are invoked on Cronus objects. There are three basic message types in this protocol: Request, Reply, and InProgress. All of the messages in the OOP are marked as belonging to the operation protocol, and each is marked with its basic type. Messages arising from one Request normally contain the same Cronus unique number called the operation identifier. A Request message also contains the operation name and a Reply message contains a standard reply code. These are the minimal contents of the messages; they also contain additional, operation-specific information.

The simplest message pattern involves one Request message generated by a client, and one Reply generated by an object manager in response.

During a manager's handling of the request, it may send an InProgress message to the original requestor. Any number of InProgress messages may be generated by manager processes handling a request; they are all addressed to the process which initiated the Request message. A client may use these messages to reset time-outs, for example.

We distinguish between a simple operation (or operation) and a compound operation. A simple operation has a single operation name and operation identifier. Any manager process, in the course of acting upon a Request may invoke one or more new (simple) operations by sending Request messages. A compound operation is the aggregate of all simple operations arising from or caused by the invocation of one simple operation. Normally, all of the suboperations will complete before the intiating simple operation completes. Each of the simple operations has its own operation identifier, so a process may invoke several sub-operations in parallel.

Sometimes a manager cannot complete the processing required for an operation; for example, a request for a catalog lookup may be satisfied only by the cooperation of catalog managers on two hosts. The manager may then either:

- perform as much processing it can, and send a Reply that is marked Incomplete, or

- elect to complete it using sub-operations, which follow the same pattern as requests, and send a Reply when the operation is complete.

If the manager chooses the first of these alternatives, it can often send the text of the message that the client needs to send to the other manager as part of the Reply. The client can complete the operation by invoking another simple operation.

It is desirable for a Cronus process to be able to query the status of a compound operation. The operation identifier of the original request is used as a global identifier for each suboperation. Since this identifier is included in the Request messages of all simple operations it causes, the managers acting on suboperations can respond to a status query keyed to the initiating identifier.

6.6 Message Structure

The primary design goal for the Cronus message structure is the regularization of control traffic. Control traffic includes requests for operations to be performed on objects, replies generated by operations, exception notices, and messages needed to coordinate distributed object managers. Control messages are usually short (tens to hundreds of octets). Because performance is a major issue, messages should be compact, and efficiently composed and parsed.

A message structure can be evaluated in a number of ways. A discussion of evaluation criteria, and an application of these criteria to a number of well-known message structures may be found in [BBN 5261]. As a result of that analysis, a standard Cronus message structure was formulated. It has the following characteristics:

- Messages are self-describing, so the fields may be identified by name rather than by order. This simplifies the parsing of messages, at the cost of transmitting the identifying information.

- The conventions rely only on features that are available

in many programming languages. This improves the portability of the implementation, at the cost of increasing the cost of a single implementation.

- The need to define new data types, which are treated in the same way as the pre-defined types, is explicitly recognized. This is consistent with the general philosophy of Cronus design.

- Name and data type fields are compactly coded, and efficient programming interfaces are provided, while the overhead of a general message format is held down. These all contribute to good system performance.

The Message Structure Library (MSL) is a collection of functions that is part of the PSL; these routines fall into three classes.

- application interface functions,

- data translation functions, and

- structure manipulation functions.

The application interface procedures construct the message in an external representation, which is machine independent, using the data translation and structure manipulation functions. This data structure can be transmitted from one process to another, and subsequently parsed by MSL procedures at the receiving process. A summary of the functions and a cross reference to detailed discussions of them may be found in Cronus User's Manual, on page msl(2).

The Cronus external representation is based on key-value pairs, where the key is a conventional name that is stored with each data value. The key indicates the meaning of the value. The value, in turn, consists of a data type indicator and the actual data. Including the type indicator assures us that we can move the data from one Cronus host to another. The internal representation of the data may differ at the sending and receiving hosts, but it is always transmitted in a canonical form, along with its type [Herlihy 1982].

A canonical type is either an atomic or composite type. An atomic type, such as boolean or signed 16-bit integer, defines a set of primitive data values. A composite type, such as array, has substructure defined in terms of other canonical types (see Cronus User's Manual can_types(4)).

Keys are coded as short (16-bit) integers, but values can vary in length from one octet to many thousands, and are not restricted in form, and may be built from simple or composite data types.

Most IPC messages passed among managers or between processes and managers use a high-level protocol called the Object-Operation Protocol (OOP). OOP is based on a set of well-known keys which are used object managers (see Cronus User's Manual keys(4))

7 Authentication, Access Control, and Security

7.1 Introduction

The goals of the Authentication and Access Control facility are:

1. Prevention of unauthorized use of Cronus and unauthorized access to DOS maintained data and services.

2. Preservation of the integrity of the system and its components against intentional insertion of unauthorized components.

3. Support for a uniform user view of access control to the resources and functions provided by Cronus

4. Survivable authentication functionality

The design of the access control and authentication facility assumes that systems in a Cronus cluster are all in a single administrative domain. There are a three broad classes of hosts within the cluster.

- hosts dedicated entirely to Cronus system functions and not user programmable.

- hosts supporting user applications using tamper-proof multiple protection domains (trusted multi-access hosts); and

- hosts supporting user applications without secure multiple protection domains (single-user workstation hosts).

We assume all hosts supporting dedicated Cronus functions and multiple user protection domains are physically secure from tampering. Workstations may not be completely physically secure, but have at least a tamper-proof component. At minimum, this component is in the local network address insertion and reception function. It could, however, be higher up in the workstation system, in the virtual local network internet address insertion and reception function, in the object system process-unique

identifier insertion and reception function, or even higher. In this sense, all user-programmable hosts support multiple protection domains (user and system), although in the limiting case, the "system" domain may simply be a piece of network interface hardware. Since we are not aware of any workstation systems meeting this requirement, we assume future product packaging changes. There seem to be two viable positions to take regarding the assumptions on these changes.

1. Assume only an absolute minimum, that a single low level "address" can be protected.

2. Allow the set of protected functions to grow as needed to conveniently interface the workstation in a manner as similar as possible to multi-access systems.

The extreme solution to the second approach could be an access machine for each workstation, although other solutions are also possible. For our current work we will assume the second approach, planning only for an arguably insecure implementation directly within the workstation.

The network (cable) itself may also not be totally physically secure. While parts of it can be expected to be secure (e.g. within a secure machine room), other parts can be expected to be exposed to unauthorized connection.

7.2 The Cronus Access Control Concept

7.2.1 Decomposition of the Access Control Problem

The basis of access control in Cronus is the ability of Cronus to reliably deliver the address of a sender of a message (or invoker of an operation) to the receiver of the message. The Cronus communication subsystem is implemented so that this is true. That is:

for IP and Virtual Local Network:

> If the sender is within the Cronus cluster, the internet host address of the sender is reliably delivered to the receiver. If the sender is not within

the cluster, a non-cluster internet host address is delivered to the receiver, which can be interpreted by the receiver as indication that the authenticity of the sender's address might be suspect.

for the Cronus IPC/object system:

> The UID of the sending or invoking process is reliably delivered to the recipient of the message.

The recipient of a request can decide on the basis of the sender's identity whether or not to perform an operation requested.

For this to be a useful basis for access control, a means for reliably associating some authorization with senders' addresses and process UIDs is required.

One approach is to make static bindings between authorizations and addresses or UIDs. These bindings would be "well-known", such that when a process receives a request from the process with UID_Y it knows that the process is acting under the Z_Authority. This method is used in the ARPANET TELNET and FTP protocols, users assume that the process for sockets one and three are under the authority of the host administration and can be trusted with their passwords. Static bindings are too restrictive to be the sole mechanism in a system like Cronus, although a few static bindings are required for the access control mechanism to work (see Section 7.6).

Dynamic binding is useful when authorities are not all known at system creation time, and when processes are dynamically created. The system must not only support mechanisms to dynamically establish the binding between a process and an authority, but also to dynamically determine the binding from some system entity in a trustworthy manner.

Most Cronus activity is the result of requests initiated by users of the system. Human users are represented by an abstraction called a "principal". If we extend the notion of a principal to include elements of the system, such as object managers, all activity in the system can be thought of as initiated by principals. System elements which are principals are called "system principals". Each Cronus principal (human or

system entity) has a unique identifier. Different system principals have different authorities. For example the primal file manager and the printer service are Cronus system principals, neither of which need be authorized for all of the objects and operations accessible to the other.

Access control can be thought of as consisting of the following steps:

1. Identification. Determine the identity of the principal that is requesting a particular operation.

2 Authorization. Determine whether the principal has been authorized to perform the operation.

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

7.2.2 Authorization

Cronus uses access control lists to support authorization. The access control list (ACL), which is part of the object descriptor, "protects" a particular action. In the simplest case, it is a list of the principals who have authorization to perform the action. When a principal attempts an operation, the list is checked for the principal, if the principal is present the authority to perform the operation has been verified and the operation may occur.

In Cronus this simple idea is extended in two ways:

1. Group identifiers may appear on an ACL, so an entire group of principals can be authorized as a unit, or have its authorization revoked as a unit.

2. A set of rights is associated with each identifier on an ACL. A single list can selectively control a principal's or a group's access to an object for which several

operations are defined, such as a file. Rights are abstract, bound to specific operations by the implementer.

An ACL is a list which contains elements of the form.

(id, rights)

where "id" is either a principal (PID) or a group identifier (GID), and "rights" define the principal's or group's authorization with respect to the object the ACL protects. The allowable rights for a particular ACL are dependent upon the type of object being protected.

Users log into Cronus as principals by supplying an appropriate name and corresponding password(10). A system component called the Authentication Manager maintains records of all principals and groups. Collectively, these records form a User Data Base (UDB). At login time the Authentication Manager expands the membership of a user-specified subset of the access control groups which he is a member. This is a transitive closure computation on the specified list of group identifiers in the user's record. The user's own id, PID, is added to the result of the expansion. The resulting set of principals is called the access group set (AGS) for the process.(11)

AGS = {PID} U Show_Group_Membership_Expanded (GID)
for the default GIDs in the PID record.

The AGS is used in access control checks as follows. When an action protected by an ACL is attempted, the ACL is compared with the principal's AGS. If an entry of the form:

(ID, (..., Right, ...))

where

---

(10). See Appendix A for a more complete description of the login and session initiation scenarios.

(11) The basic ideas associated with Access Group Sets have been adapted from similar work at Carnegie Mellon University in the Central File System project.

ID is in AGS, and
Right is required to perform the action

is found on the ACL, the principal's authorization is verified and the action may be performed.

During a session, a user may add and remove identities from the current AGS. To add a group identity, the user must be a member of the added group. Updating the current AGS is accomplished via operations invoked on the Authentication Manager, which causes the update of the current process AGS list. These operations affect a single process however, the new AGS will be inherited by subsequently-created children only.

7.2.3 Identification in Cronus

There are two related identification problems.

I1. At the start of each session, the identity of the user must be established.

I2. Processes must be able to ascertain the identity of the principal corresponding to the processes with which they interact.

The solution to both problems lies in a set of mechanisms that bind processes with principal ids and group identifiers. These mechanisms depend upon the ability of the communication system to deliver the UID of a sending process to the receiver of a message reliably.

It is useful to restate these problems into the following terms.

1. A binding must be established between a process and an AGS.

2. There must be a means for a process P1 to determine the binding between another process P2 and its AGS.

When a user approaches Cronus to start a session a process (P1) is allocated(12). P1 cannot be bound to U (the user's principal identifier) until Cronus establishes the connection via password authentication. Before that happens, P1 is bound to a well-known principal, "NotLoggedIn", which has minimal authorization. One task of the login procedure is to change the binding of P1 from NotLoggedIn to U.

The binding between a principal identity and a process is established by the Authenticate_As operation. The user engages in an authentication dialogue with Cronus, supplying a name and password which is checked against the UDB. If the authentication dialogue succeeds, the AGS for U is computed and a binding is established between P1 and U. A record of the binding

P1, U, AGS

is maintained by the process manager for the authenticated process, to be used throughout the process lifetime. The identity of the user has been established, completing problem I1.

Throughout the course of U's session, P1 and other processes acting on behalf of U attempt actions which require authorization verification by the processes that perform the actions. This is problem I2. Consider a situation in which P1 has requested another process (S1) to perform some action (A), shown in Figure 1.

In order to perform an access control check, S1 needs to determine the binding of P1. The identity of P1 is known to S1 because P1's UID was delivered along with the operation invocation that requests A. S1 can obtain the binding of P1 by invoking the Authorization_Binding_Of operation.

Authorization_Binding_Of(P1) -> U, AGS.

Authorization_Binding_Of causes a message to be sent from S1 to

---

(12). Cronus actually uses a more complex process structure to support a user session, as indicated in Appendix A.3. However, the following discussion is insensitive to these details, so we use this simple model in our explanation

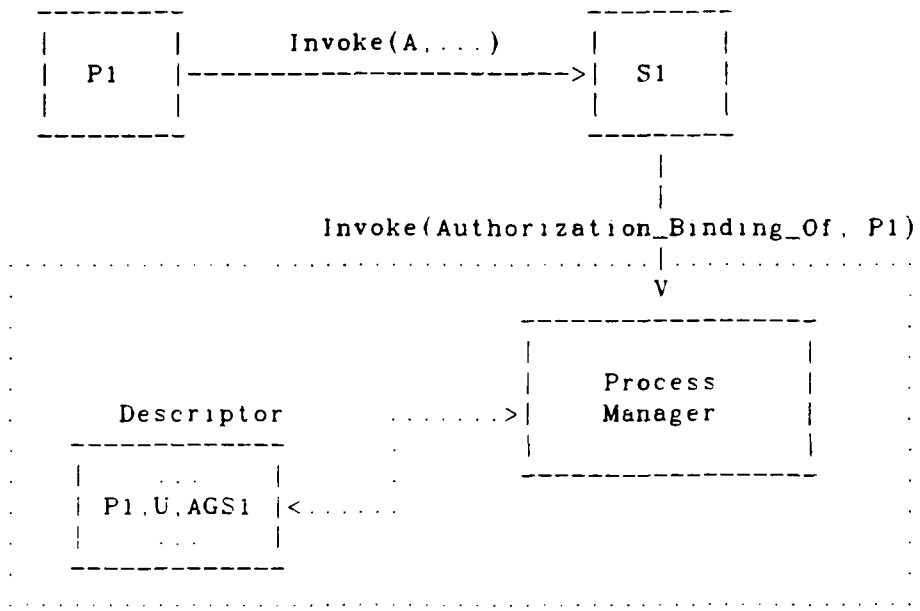


Figure 7.1 Retrieving Access Control Data

the manager for process P1, which returns the bindings for the process to S1

The login sequence establishes a binding between user (U) and an "initial" process (P1). Bindings are established for other processes created during a user session through inheritance. During a user session, processes created by an authenticated process inherit both the principal identity and the current AGS of the initiating process. Object managers attain their principal identities and access group sets as part of the system initialization phase.

7.3 Access Control List Initialization

A common problem associated with Access Control List mechanisms is the effort required for proper explicit (manual) initialization. In practice, the ACL for a new object can often be automatically predetermined based upon the type of the object, the creator, and the context in which the object is created (primarily the directory in which it is subsequently catalogued). This is the premise upon which the Cronus Initial Access Control List (IACL) mechanism is based.

A list of type-specific IACLs may be associated with selected Cronus objects, currently Principal and Directory objects. The IACLs are manipulated using the standard ACL manipulation operations (ReadACL, AddToACL, RemoveFromACL), distinguished by an optional key denoting the type with which the IACL is to be associated. The IACL mechanism also supports the Cronus type hierarchy. the IACL associated with an ancestor in the type hierarchy will be used if a more specific IACL for the type itself has not been specified.

Cronus Create operations incorporate the following algorithm for initializing the ACL of newly-created objects.

1) A list of "IACL hints" (UIDs of objects potentially having IACLs associated with them) are searched in order for an IACL pertaining to the type of the object being created. The first one found is used. These hints usually reference the Cronus directory where the object will subsequently be catalogued.

2) If no IACL search is specified, or the hints fail to yield an appropriate IACL, the object for the Principal invoking the operation is queried as if it were included at the end of the hints list.

3) If an IACL is still not found, the invoking Principal is given all rights to the object.

There are user commands for setting up, examining and modifying the initial access control lists retained with cronus objects.

7.4 Authentication Manager

The Authentication Manager defines and maintains two types of abstract Cronus objects: CT_Principal and CT_Group. Like other system objects, the CT_Principal and CT_Group identifier objects have symbolic names for convenient human access. Principals are symbolically named from a private name space maintained by the Authentication Manager, which ensures their uniqueness across the entire system. Symbolic group identifiers can be placed anywhere in the Cronus catalog, at the convenience of the creating user.

Operations on objects of type CT_Principal and of type CT_Group are controlled by access control lists. By convention, any legitimate principal can create a new CT_Group object, but only administratively authorized principals can create a new principal. When the system is initialized, it contains at least one pre-defined principal, which is authorized to create other principals.

In the following sections we discuss the design ofthe objects and operations supported by the Authentication Manager. Section 7.8 discusses how to make the functions of the Authentication Manager survivable.

7.5 Objects Related to Authorization

The object of type CT_Authentication_Data is the user data base consisting of the records for system users and for groups of principals which have been defined in the system.

The object of type CT_Principal is the permanent data base entry that Cronus maintains for each legitimate user. It is the repository for such user-specific data as default priority and other parameters associated with resource management; default modes of behavior (e.g. default working directory); and authorization data. It is expected that new kinds of data will be added to the principal objects from time to time.

A CT_Principal object can be expected to contain the following data:

- Principal unique-identifier (PID)
- Symbolic name of principal
- Access control list
- Encrypted password
- Direct group memberships
- Direct group memberships to be expanded on Login
- Range of priority service authorized
- Default priority
- Name of default initial subsystem
- Name of home directory for the principal ... (other user-specific data)

The priority data will be used in resource management functions. The default subsystem is the program automatically invoked following login. A home directory is a directory assigned to the principal that serves as the initial current directory for catalog accesses; in particular, it contains additional user initialization data.

Groups (objects of type CT_Group) gather a number of identities for purposes of collectively granting them rights to objects and operations. Any user can create a new group, and place any other principal or group in it. This group can then be placed on an ACL. The access control list for the group object controls modification of the group definition.

A CT_Group object contains at least the following data:

- GID for the group
- Name of the group
- GIDs of the groups of which the group is directly a member
- IDs of principals (PIDs) and groups (GIDs) that are direct members of the group

There are a few special group identifiers. One of these (group world) represents the set of principal identifiers without actually enumerating them anywhere. This group identifier is automatically appended to every AGS computation. Another special

group "Wheel" represents an access control override capability used for system maintenance, implicitly receiving all rights to all Cronus objects. Admission to this group is carefully controlled.

A convention has been adopted which effectively supports wheel capability only for objects of a specified type. A process whose principal ID matches the PID of the manager process is automatically granted all rights to all objects managed by that manager. This is useful in handling peer managers. As an example, all file managers are bound to a special file manager principal, and implicitly have all access to all files managed by peer file managers.

7.6 Operations on Authorization Related Objects

The generic operations to create and remove objects, and to examine and modify the object descriptor, ACL, and object status apply to instances of CT_Principal and CT_Group.

The following operation (see Cronus User's Manual auth_data(3)) is used during login to establish the binding of the user to the principal UID:

Authenticate_As

The following operations allow processes to control the identities applicable to an authenticated process (see Cronus User's Manual auth_data(3)). They effect only a single process, which may be either the invoking process or another process authenticated to the same principal.

Enable_Access_Group
Disable_Access_Group

The following operations maintain and interrogate the objects of type CT_Principal (Cronus User's Manual principal(3)).

Lookup_Principal
Show_Group_Memberships

Add_to_Default_Group_Expansion_List
Delete_from_Default_Group_Expansion_List
Change_Password

The rest of the data in the principal entry in the user data base is treated as part of the object descriptor. The generic operations which manipulate the object descriptor are used to examine and set these fields.

The following operations are used to inspect and maintain the group identifier objects (Cronus User's Manual group(3)):

Add_to_Group
Remove_from_Group
Show_Group_Members

The rest of the data in objects of type CT_Group is contained in the process descriptor and is maintained using the generic operations defined on object descriptors.

The access control list of any object, including objects of type CT_Group and CT_Principal, can be set using the generic operations on access control lists (see Cronus User's Manual object(3)).

7.7 Operation of the Access Control Authorization Function

Cronus access control checks the current identity of the accessing agent against access control lists maintained by the service provider. A process is authenticated in a way which binds the process UID to a set of external identities defining the authorizations of the process. These identities, the AGS, are available to any service-providing process. This section discusses the authorization function which is part of the service provider.

In general, the access control steps within an object proceed as follows:

1. The request is parsed to determine the originating process UID and the operation/object requested. The process_UID is trusted because it is added to the message by the operation switch. Universal public privilege for the operation to all objects managed by the manager is first checked, to see if the specific access check is needed.

2. A manager-based cache of process/object authorization pairs for the process_UID is checked for a valid current entry.

3. If there is no corresponding cache entry, the accessing agent's AGS is obtained. This data is also cached but on a per-host basis by the AGS cache manager. If present on the host, this cache manager provides a high performance interface to the Authentication_Bindings_Of function. There is a broadcast-based protocol for alerting AGS cache managers to entries that should be purged. If an AGS cache manager does not run on a host, managers execute the Authentication_Bindings_Of operation directly, and the AGS is not cached. [The per host AGS caching is not yet designed or implemented.]

4. The access control software computes a new process_UID/object authorization entry using the AGS and the access control list maintained with the protected object/operation. The process_UID authorization entry is then put in the manager cache.

5. The process UID object authorization is used to verify permission. If authorized, the operation is passed on to the operation code. If unauthorized, the request is rejected.

6. To allow for the enabling of new access groups, steps 3-5 are repeated in the event that cached AGS fails.

The permission authorization function is accomplished by a set of routines and data structures that we call the "gatekeeper" because of its role as protector of the objects/operations. Gatekeeper functions can be invoked as part of the procedures for receipt of a message, or called directly from the host process.

Access control can be applied to operations on the object set supported by the receiving manager process, or on operations defined by the receiving service. There is a fixed maximum number of access control rights maintained by the gatekeeper software (currently 32) for any object. These rights are represented as positions in a bit vector associated with both the identity it authorizes (principal identifier or group identifier) and the object it controls.

7.8 Host Registration

The lack of physical security for various parts of the system presents problems for the access control subsystem. Since the network cable may be accessible to tampering, the network might be tapped. An outsider could then inject or inspect packets under an assumed network address. A workstation might pose as the site of a trusted manager. We can use administrative authorization to alleviate these problems.

Encryption of all local network traffic is a form of authorization. It can remove the threat of tapping for either listening for or insertion of packets. Providing the host with the encryption/decryption key is administrative authorization to participate in the Cronus cluster. If a host can communicate at all, it can be considered an authorized host. Because encryption/decryption is isolated in the communication interface, it can be added transparently at any time. While communication encryption can be thought of as part of the Cronus design, it will not be part of the initial implementation.

Since workstations may be treated specially for some access control decisions, system configuration registry could be the source of such identification. In addition, the undesirability of tightly controlling responses to broadcast Locate operations, makes the registry useful in determining the authenticity of the respondee. A configuration registry enumerates all of the authorized system hosts, and the system services (Cronus functions) which they have been authorized to run.

One secure way to make the registry service available is to support it on one (or more) well-known Cronus hosts (i.e. hosts at a well-known internet addresses, say host No. 1, ...). The

configuration data can then be obtained with an Invoke On Host to the well-known hosts using the logical name for the service(13). The cluster configuration service would support the following functions:

Show_Configuration_Hosts
Set_Configuration_Hosts

Standard access controls apply, with Show_Configuration_Hosts being universally allowed, while Set_Configuration_Hosts limited to a system administration group.

7.9 Survivable Authorization Design

7.9.1 Objectives

The authentication function and evaluation of the current AGS are critical parts of the operation of Cronus. These functions must be available at all times or Cronus cannot operate effectively. Our objectives in providing survivability in Authentication are:

a. A Cronus user should, under reasonable failure patterns, always be able to gain access to the system.

b. The current value of the process-AGS binding should be available whenever a process is able to request services from object managers.

c. A less important but desireable objective is that a client be able to continue to perform maintenance operations on the principal and group objects despite failures of hosts supporting these functions.

To meet objectives (a) and (c), we must replicate the Authentication function. To meet objective (b), we must maintain

(13). Since this function is often used to determine the veracity of responses to the Locate operations, it can not safely use Locate to find out where configuration managers are running.

the bindings in a replicated fashion, or keep them close to the process to which they refer, so that the bindings are available when the process makes requests of other Cronus managers.

7.9.2 Observations

The authentication function is a global DOS function supported on a GCE which is expected to be up most of the time. Because these services are simple, the host hardware and software should be stable, increasing its availability. Since the GCE is relatively inexpensive, it is also feasible to stock a spare.

The authentication function is based on maintaining two related types of objects. The data bases which the Authentication Manager maintains to support the principal and group objects are not large. The principal data base is estimated to be no larger than 1000 users, with an average entry having around 1000 bytes of data. The group data base might have 2000 entries, averaging 300 bytes of data. This is less than 2 MBytes of data, and can easily be accommodated on a GCE.

The processing demand on Authentication managers is not expected to be large. Aside from initial authentication and group expansion, which occurs typically once per user per session, other operations are infrequent. New users and groups are occasionally created and the associated data bases occasionally displayed and updated. A single GCE appears easily capable of handling anticipated processing requests.

Performance and size considerations do not seem to require more than a single GCE per cluster. Survivability is the primary motivation for replicating the authentication manager. Our approach is to maintain completely replicated data bases on two or more GCEs.

Of the operations performed by the Authentication Manager, the one of most concern for survivability is Authenticate_As, which is a read-only function. This is also true of a number of other AM operations (Lookup Principal, Show Groups Expanded, etc.). Synchronization of multiple authentication managers is not required to complete these operations.

Some AM operations do modify the authentication data (e.g. Create new principal, Modify User Parameters, etc.). These require synchronization among Authentication Managers for consistency. However, because these operations are relatively infrequent and have simple semantics, a simple approach to synchronization which ignores maximizing concurrency will suffice. We designate a primary Authentication Manager as a single point of synchronization. This method is backed up by an alternate procedure if the primary site is inaccessible. A complete description of our approach follows in the next section.

In the current implementation, each process has a process manager on the same host. The process-AGS bindings are maintained by the process manager in the process descriptors for these processes. During host outages when a manager is inaccessible, so too will be the process it manages. There is no need to maintain the process-AGS binding any more reliably than we maintain the process reliability. As some later point, we will address issues of process survivability. We can then naturally think in terms of replication of process descriptor data (including the current AGS) as part of the reliable process concept, and need not address it separately.

7.9.3 Approach

Fully redundant copies of the authentication data bases are maintained at more than one Cronus host. This means that, ignoring synchronization, an operation can be completed at any site which maintains the data base. We expect that two operational authentication sites will provide sufficient availability for most applications of Cronus.

A spare GCE could be integrated into the system if one of the dedicated hosts needs to be taken off-line for any extended period. This minimizes the time during which there may only be a single Authentication site functioning. The new host integration protocol first involves transmission of all of the existing objects. When the object transmission is complete, the new manager retrieves the change log and incorporates any updates. The final step before assuming operational status is to coordinate with any on-going activities.

Each operation on authentication data objects is an independent transaction, so that there is no linkage between any two operations. The operations either reference the identified objects (read operations) or modify the identified objects (write operations). Read operations require no synchronization or concurrency control between Authentication Managers. Any Read operation can be handled by any available authentication manager. Some read operations have side effects which do change the state of other system variables (e.g. Authenticate_As modifies the current process AGS in its process descriptor) but these are idempotent operations so repeating them at distinct sites as part of error recovery is not harmful.

Write operations, on the other hand, require synchronization among the Authentication managers to preserve the consistency of the data with respect to concurrent updates. To do this one AM is chosen as the primary site. The designation of which AM is primary is found in the configuration data base for the system. Clients as well as other AM processes can consult this data base to find the primary site. The primary site remembers its role and will respond to broadcast request to identify itself in case the configuration file is inaccessible.

All Write operations are initiated with the Primary AM, which serializes the modifications to the database. The primary AM records the modification in a change log by appending a change record to a multi-copy reliable file. After logging the request, it updates it own data base, and informs other operational AMs of the change. If all AMs are running, the data bases are again synchronized after each one incorporates the update. When an AM is restarted, it processes the change log to incorporate changes made to the data base in its absence before it will accept new requests. Multi-copy files are used for change logs to avoid single host failure reintegration dependencies.

This approach raises two issues.

a. What, if anything, should we do about read/write synchronization for read operations that may be processed by a non-primary AM while the corresponding object is undergoing modification by the Primary AM?

b. What, if anything, should we do when a modification is requested and the primary AM is inaccessible?

To answer question (a) we first observe that not only is the data changed infrequently, but much of it is particular to a single Cronus user, and hence concurrent read and write access is quite unlikely. Furthermore an old copy of just modified data is almost never harmful. The behavior is similar to a race condition between independent accesses to a single copy data base. Thus our approach to Read/Write synchronization is to do nothing.

There are many possible answers to question (b). One approach is to do nothing, and reject these operations temporarily until the primary AM is brought back on-line. Since modifications to authentication data are not critical to the operation of the system, the major effect of this is inconvenience because we will need to repeat the operations at a later time. A simple mechanism which avoids this uses the lock on the change log file as a tool for serializing updates from any of the available AMs. In this scheme, when the primary AM is inaccessible, any AM can initiate the update if it can first lock the change log. It then informs the other operational AMs of the change. When the primary comes back, it integrates the changes it has missed before assuming primary update responsibility again.

8 Cronus File System

8.1 File System Overview

Cronus supports a number of different kinds of files, including.

- o Primal files.

  The primal file is the most basic kind of Cronus file. Other kinds of Cronus files are implemented from primal files. A primal file is stored entirely within a single host, and is bound to the host.

- o Reliable files.

  A reliable file is 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 derives from the fact that the file is accessible as long as at least one of the primal files that implement it is.

- o Dispersed files.

  A dispersed file is implemented by one or more primal files. A dispersed file is one whose contents may be distributed over more than one host. Each of the primal files used to implement a dispersed file contains part of the contents.

The initial Cronus implementation (the "primal system") supports only primal files, which are implemented upon underlying single-host file systems. The next major Cronus release (the "reliable system") will support reliable files. Later system releases may support dispersed files.

This section also describes a single host file system, called the Elementary File System, which will be developed for each Cronus file host to serve as a common base of implementation support for Cronus file managers.

Primal files are Cronus objects. They have unique identifiers (UIDs), and may be given symbolic names. There is a Cronus object type CT_Primal_File.

Executable programs will be stored as files of type CT_Executable_File which is a subtype of primal File. There will be many different kinds of hosts in Cronus, and an executable program file which can run on one host type will usually not be able to run on another. In addition to the normal descriptive information, files of this type have information that specifies where they can be run. The additional information maintained for an executable file would include:

- The type of processor required to execute the program stored in the file.
- The run-time environment required by the program including the local operating system and necessary peripheral devices.

8.2 Cronus Primal Files

8.2.1 Cronus Primal Files

Primal files cannot be moved from one host to another; the primal file system is partitioned among hosts that store primal files. The HostNumber component of the UID for a primal file always specifies the host on which the file is stored. A copy of a primal file can be created on another host, and the original can be deleted. The copy is a different primal file with a different UID, it just happens to contain the same data as the original file.

Like other Cronus objects, primal files are accessible to processes by means of the interprocess communication and operation switch (Section 6). There is a Primal File Manager process on each host that stores part of the primal file system. A client process accesses a primal file by invoking an operation on the file, in which the UID for the file and the operation to be performed on the file are specified.

The Primal File Manager that maintains a primal file also defines a mapping between the UID for the primal file and the information required to manage the file. The collection of information necessary to manage a primal file is called its descriptor. The file descriptor includes.

UID of the creator;
Date and time of creation,
Date and time of last write.
Access control list (ACL) for the file;
Information necessary to find the file data on the storage media;
Current size of the file,
Other information (to be specified as needed)

...

Most of the operations provided by conventional file systems (create, read, write, etc.) are implemented for Cronus primal files. The design is discussed in terms of the normal life cycle of a primal file which includes.

1. The file is created.
2. Data in the file may be read or written by a client.
3. Information in the file descriptor may be read or written by a client.
4. The right to access the file may be granted to or revoked from other users.
5. The file may be deleted.

File creation involves: the generation of a UID; the creation and initialization of a descriptor for the file; and the binding of the UID and the file descriptor in the Primal File UID Table. Until data is written into the file, the file is empty. When a primal file is created by a Primal File Manager, it is created on that manager's host.

There is an issue regarding whether it should be necessary to open a primal file before reading or writing file data. One reason for "open" and "close" is to provide for reader-writer synchronization, another is optimization of read/write operations. The disadvantage is that open/close add complexity to the Primal File Manager because it must maintain state

information for open files and deal with the problem of files opened which are never explicitly closed (e.g., because the client's host has crashed). Furthermore, if we require open and close, additional operations must be invoked on the file even when the read or write is for a small amount of data.

The Primal File Manager supports access to files without open and provides an open/close facility for clients that need it. A read or write without open is called a "free read" or a "free write". The client may then choose whether the additional overhead of opening and closing the file is worthwhile. For example, if we wish to write a simple log message when a process is initiated, we would probably choose the free write. If, on the other hand, we were copying a file, we would probably choose to open the files, incurring the overhead of initiation once, and gaining further system support for synchronization and data integrity. A client process may read or write data in a primal file (subject to authorization considerations) without opening it, unless another process has opened the file in such a way that free reads and writes are forbidden.

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

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".

Thus, the access states defined for a file are:

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

A file may be opened so long as the access state requested does not conflict with the current access state of the file. Table 6.1 defines the compatibility of the access states with one another, and with read and write operations invoked by a client without previously opening the file. 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.

The data in a primal file is a sequence of octets, numbered from 0 to N. The read operation specifies the first octet to be read and the number of octets to be read. The write operation specifies the octet position of the first octet to be written and N octets of data to be written.

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

A file is open to a process. The Primal File Manager provides an operation which returns a list of the UIDs for the processes, if any, that have a given file open. Another operation returns a list of the UIDs for the files, if any, that a given process has open.

When a process is destroyed with files open, the files are

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

Table 8.1 Access State Compatibility

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.

A client can read the descriptor of a primal file. Some of the information in the file descriptor is changed as a side effect of operations on the file. For example, when a file is written, the date and time of last write is changed. There is other information that the client may wish to change explicitly.

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.

Some file system support the notion of Delete, UnDelete and

Expunge operations. The current design for the primal file system assumes that only Delete (called Remove) will be supported, but it is relatively straightforward to modify the specification of Cronus primal files to accommodate a Delete, Undelete, and Expunge model of file removal.

8.2.2 Crash Recovery Properties

If a primal file operation is invoked, the Primal File Manager normally acknowledges the operation, indicating the disposition of the operation (e.g., success, failure, and reason) and, depending upon the operation, to return any data requested.

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.

This atomicity property is true for the Close-and-RetainWrites operation. That is, either none or all of the writes made while the file was open will have been performed.

8.2.3 Operations for Objects of type CT_Primal_File

In addition to the generic operations (Cronus User's Manual object(3)) the following operations are supported for primal files.

Open
Close
Sync
Read
Write
Truncate
Append
FilesOpenBy
OpenStatusOf
CloseProcessOpenFile
CloseAllProcessOpenFiles

The Open and Close operations provide an atomic transaction capability for a single primal file. At some later point, we may define explicit BeginTransaction, EndTransaction, and AddToTransaction operations which could be used to provide a capability for transactions that involve more than a single primal file.

In response to a Status operation, the Primal File Manager returns information about the status of the primal files it manages (Cronus User's Manual p_filesys(3)), such as the amount of free space, the amount of space used by existing files, the number of files it manages, the number of files currently opened, etc. This information will be useful to system operations personnel as well as to clients who might use it when deciding where to create primal files.

8.3 Reliable Files

8.3.1 Objectives

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

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

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

Furthermore, in the Cronus environment it is desirable to support concurrent access to files. For example, Cronus supports a form of multiple readers / single writer concurrency control for primal files. The same sort of concurrency control is provided for multi-image files.

Concurrency control requires that sites managing images of a file cooperate to synchronize client access to the file. There is complexity and overhead associated with this cooperation. 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 concurency 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 Cronus 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 approach to concurrency control is to 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 relaxed approach to concurrency control will be practical only if:

a. File access patterns are such that it is relatively unusual for multiple concurrent updates to occur.

b. Hosts are reasonably reliable so that host failures that prevent strong synchonization are relatively rare.

c. There is a simple and inexpensive way to detect inconsistent images of a file. We believe that the Version Vector mechanism developed at UCLA [Parker 1983] is a good one for this purpose.

Experience with Cronus may show that there are some applications which require more absolute synchronization than this approach supports. If that proves to be the case, the support for reliable files will be augmented to include a file type for which more positive synchronization is supported.

8.3.2 Reliable Files as Composite Objects

A reliable file is a Cronus object of type, CT_Reliable_File. 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.

A reliable file of cardinality = 1 is a migratory file. Although it has only a single image like a primal file, unlike a primal file it may be moved from one host to another.

Each Reliable File Manager (RFM) 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.

Depending on the cardinality of a particular reliable file, a RFM may need to cooperate with 0 (cardinality = 1), 1 (cardinality = 2), or more (cardinality > 2) other RFMs. For each reliable file it manages, a RFM is directly responsible for carrying out the operations on a particular primal file that represents an image of the file. We shall sometimes refer to that image as the manager's image or as the local (to the manager) image.

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.

For Cronus, 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. RFMs, of course, access the file images through PFMs in the normal fashion.

There is an issue regarding the relation of RFMs to PFMs. They could be implemented either as two completely separate managers which communicate by means of interprocess communication or as a single, combined manager for both CT_Primal_File and CT_Reliable_File. The initial implementation of reliable files will be accomplished by means of RFMs that are separate from the PFMs. Later implementations may integrate the RFM functions into (some of) the PFMs.

In addition to the information maintained in descriptors for primal files, object descriptors for reliable files contain the following information.

File Cardinality.
ID of primary site (see below).
Version vector for the local image of the file (see below).
Version vector for the local image of the descriptor (see below).
List of UID's for the primal files that implement images of the file.

8.3.3 Synchronization Considerations

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), Cronus must control and synchronize the manner in which clients access the files.

The general Cronus approach to synchronization for reliable files 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 be 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 for Cronus. Should the mechanism prove to be inadequate (for example, because it cannot achieve synchronization often enough, given the failure patterns observed in Cronus), it will 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. This locking discipline is described in more detail in the Cronus User's Manual.

The reliable file system supports the notion of free reads and writes. For a free read the synchronization outlined in Table 8.1 is performed by the file manager which handles the client's read, but no attempt to synchronize with the primary manager is made. Free write operations require synchronization with the primary manager.

If sychronization 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 the 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 (Si, Vi) counts the number of times updates to RF were initiated at Si. Thus, each time an update to RF originates at Si, Vi 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 (Si, Vi) 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.

Since the descriptor for a file may undergo modification independently of the file data, descriptors for reliable files also have version vectors.

The question of when version vectors for file images should be compared and what to do if they are not equal is discussed in Section 8.3.6. The synchronization mechanism for reliable files outlined here is described in more detail in the Cronus User's Manual.

8.3.4 Interactions Among Reliable File Managers

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.

Occassionally 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 inititiating RFM to ensure that the interaction occurs. The mechanism used by RFM's to do this is as follows.

Each RFM maintains a PendingActions data base which contains a record for each operation it was unable to completely perform due to its inability to interact with other RFM's. Each such record includes.

- the UID of the reliable file,
- a specification of the action required to complete the operation,
- a list of the sites at which the action must be performed (for some actions, this list may be empty).

Whenever the RFM is unable to complete an operation, it adds a record to the PendingActions data base to describe the actions necessary to complete the operation. Subsequently, at regular intervals, the RFM scans the PendingActions data base and for each record, it attempts to perform the necessary interactions. If the RFM succeeds in performing some, but not all, of the interactions, it updates the record. When all of the interactions described by a record are successfully performed, the record is removed from the data base.

The actions that may be found in records in the PendingActions data base include.

a. Acquire sites to store images of a file.

b. Update the descriptor for a file.

c. Update a file itself.

When a RFM comes up for the first time, its PendingActions data base is empty, and if sites and the network never failed the data base would remain empty.

The PendingActions data base should be stored in a reasonably reliable fashion. It is probably adequate to store it as a primal file on the RFM's local site.

8.3.5 Operations on Reliable Files

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).

The design of reliable files is conveniently described in terms of the normal life cycle for a file, which is much the same as that for a primal file. The principal exception is that the cardinality of the file may change. The life cycle includes:

a. The file is created.

b. Data in the file may be read by a client.

c. Data in the file may be written by a client.

d. Information in the file descriptor may be read by a client.

e. Information in the file descriptor may be written by a client.

f. The cardinality of the file may be changed.

g. The file may be deleted.

The following sections discuss these operations.

8.3.5.1 Creating Reliable Files

A reliable file must be created before data can be written into it, and until data is written into the file, the file remains empty.

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.

For the initial implementation of reliable files, clients may exercise control only over where primary file images are maintained. If the Create operation is requested by means of InvokeOnHost, then the RFM at that host becomes the primary manager, otherwise, the RFM selected by the interprocess

communication facility becomes the primary manager. Later implementations may provide means for client processes (as well as for users through the user interface) to exercise control over the initial placement of secondary images. After images are in place, the Move_Image_To_Site operation can be used to move an image from one site to another.

When a RFM receives a Create operation, it:

a. Creates a (empty) primal file for the primary image of the reliable file, and obtains its UID (UID_pf).

b. Allocates a UID (UID_rf) for the reliable file, and makes an entry for it in its UID table.

c. Creates and initializes a descriptor for the reliable file. The following descriptor fields are initialized.

   The cardinality,
   The primary site,
   The file version vector and descriptor version vector,
   The list of UIDs for images is initialized to include UID_pf.

d. Returns UID_rf to the client, indicating that the Create succeeded.

Secondary images of the file are not created until the file is written the first time. (That is, after a free write or after the file is opened, written into and closed).

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.

a. The selection of the new sites.

b. Obtaining commitments from the RFMs at the selected sites to store images of the file.

c. Updating file descriptors at each of the file sites to reflect the new sites.

The RFM acquisition procedure is structured so that an RFM need not, as part of a single acquisition attempt, acquire every site required to support a file's cardinality. An RFM can support operations on a reliable file even if not all of the desired images of the file have been created. When an RFM is unable to acquire all the sites necessary to achieve the desired file cardinality, it creates a record in its PendingActions data base to ensure that the additional sites will be acquired.

The acquisition procedure is described in more detail in the Cronus User's Manual.

8.3.5.2 Reading Reliable Files

Reading a reliable file is similar to reading a primal file. File data may be read by means of a free read operation, or by opening the file prior to performing read operations. In either case the interprocess communication facility delivers the operations to an RFM that manages the file.

There are several differences in dealing with reliable files which are visible to a client. These include the following:

a. The interaction between the RFM that receives the operation and the primary RFM for the file in order to achieve synchronization is not visible to the client. However, should the synchronization fail because the primary RFM is inaccessible, the client will be informed and given an opportunity either to continue with the access or to abort it.

b. A client process can obtain a list of the sites that have images of a reliable file, and it can choose which RFM to deal with to access the file. For example, it might choose the primary RFM, or, if an RFM happens to reside on the host it does, it might choose that one.

c. After it opens a file, the client should continue to deal with the same RFM for operations on the open file until it closes the file.

8.3.5.3 Writing Reliable Files

Writing a reliable file is similar to writing a primal file. The principal differences are essentially those noted above for reading reliable files: the required synchronization may fail due to the inaccessiblity of the primary manager for the file, in which case the client must decide whether to proceed at some risk or to abort the write, the client may choose the RFM with which it deals; and, after it has opened a reliable file for writing, a client should deal with the same RFM for operations on the open file until it closes the file.

File data must be updated after a free write or after a file opened for writing has been closed (if writes have actually been made and are to be retained).

The RFM at which the writes 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:

a. It increments its (Site, Version) element of the file version vector.

b. It attempts to interact with each other RFM that manages an image of the file.

c. Should it fail to complete the image update with any RFM, it adds a record to the PendingActions data base specifying the file and the RFMs it was unable to update.

The actual update procedure for a particular image involves several exchanges between the initiating RFM (iRFM) and the responding RFM (rRFM), and works roughly as follows:

a. iRFM does InvokeOnHost(SiteOf(rRFM), UID, UpdateImage, DVV, FVV),

where UID is the UID of the reliable file, DVV is the version vector for the file descriptor, and FVV is the version vector for the file itself.

b. rRFM compares both DVV and FVV against the descriptor and file version vectors it maintains for UID. Assuming that

DVV and FVV dominate the corresponding version vectors at rRFM. rRFM returns to iRFM a SendTheDescriptor message. (Section 8.3.6 discusses what happens if iRFM's version vectors are dominated by or are incompatible with rRFM's.)

c. When iRFM receives the SendTheDescriptor message, it sends the new value of the file descriptor to rRFM in a HereIsTheDescriptor message.

d. rRFM receives the file descriptor and updates its copy of the descriptor. It then returns iRFM a SendTheFileUpdate message.

e. When iRFM receives the SendTheFileUpdate message, it transmits the file update to rRFM in a HereIsTheFileUpdate message. Depending on the nature of the changes to be made to the file image, the update may be transmitted by sending the entire file or by sending only the changes that need to be made to the file to update it.

f. Finally, after it has stored the new file data in the primal file that holds its image of the file, rRFM returns an UpdateImageSucceeded message to iRFM.

8.3.5.4 Other Operations

This section describes the Change_Cardinality and Move_Image_To_Site operations. Both operations require synchronization with the primary manager.

Change_Cardinality is used to change the number of images the system tries to maintain for a reliable file. An increase to the cardinality is accomplished by execution of the acquisition procedure described in Section 8.3.5.1. Decreasing the cardinality is roughly the inverse of increasing it. The performing manager selects a site or a set of sites which currently maintain images of the file and asks the manager at each to agree to discard its image of the file, and to remove the

file from its UID table. After each agrees, the performing manager instructs each to discard the image and the remaining managers to update their descriptors for the file.

Move_Image_To_Site moves a file image from one site to another, preserving the file cardinality. The parameters of the operation are the file UID, the site of the image to move, and a new site to hold the image. The operation involves creating an image of the file at the new site, discarding the image at the old site, and updating the descriptors held by all managers of the file to reflect the change.

8.3.6 Use of Version Vectors

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.

a. 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.

b. 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.

In each situation, both the descriptor version vector and the file data version vector are compared. There are four possible outcomes for the comparison of version vectors:

a. The supplied version vector is the same as the local version vector.

b. The supplied version vector dominates the local version vector.

c. The supplied version vector is dominated by the local version vector.

d. The two version vectors are incompatible.

The actions taken for these outcomes depend upon whether image updating or file locking is taking place.

For updating, the version vectors are compared by the RFM whose image is about to be updated. The various comparison outcomes and the actions to be taken for each are.

a. The supplied version vector is the same as the local version vector. Since the updating RFM increments its element of the version vector prior to sending it for comparison, if the RFMs are behaving properly, this case should not occur. If it does, some RFM has been misbehaving. The update should be deferred and the operations staff should be alerted by means of a message to the Monitoring and Control System.

b The supplied version vector dominates the local version vector. This is the normal case, since the local image is being updated. In this case, the image update should proceed.

c. The supplied version vector is dominated by the local version vector. In this case, the local image is more recent than the one that is to replace it. The update should be aborted, and the local version should be used to update the remote version.

d. The version vectors are incompatible. This detects an inconsistency. The update should be deferred until human intervention can clear up the problem.

In the locking situation, the version vectors are being compared by the primary RFM for the file in question.

a. The supplied version vector is the same as the local version vector. This should be the normal case, and locking can proceed.

b. The supplied version vector dominates the local version vector. In this case, the primary image is obsolete, and should be brought up to date. If the file is being locked for writing, the locking should proceed, and the local image can be updated when the file is closed. If the file is being locked for reading, there are two possibilities. Either, the primary file image could be updated before proceeding with the locking, or the locking could proceed and the file could be updated when the lock is cleared.

c. The supplied version vector is dominated by the local version vector. The secondary image should be updated before proceeding. If the file is being locked for reading, then the file image at the secondary site should be updated so that the client is given access to the most current file data. If the file is being locked for writing, then the secondary file image must be updated first to avoid incompatibility.

d The version vectors are incompatible. If the file is being locked for reading, the locking may proceed, but an attempt to signal a user or operator to resolve the incompatibility should be made. If the file is being locked for writing, the client should be informed of the incompatibility and given an opportunity to resolve it. The client may proceed without resolving the incompatibility, in which case the write is treated as an unsynchronized write.

8.4 Elementary File System

8.4.1 Introduction

The Elementary File System (EFS) is an easily ported single host file system that serves as a common base of implementation support for Cronus file managers on Cronus Generic Computing Elements (GCEs) configured with disks, on the UNIX system, and on the VAX. The underlying implementation of the EFS is consituent host dependent, but the interface presented to the Cronus File Manger is uniform. As a result, portability of the File Manager

is enhanced, and the cost of integration of new hosts is reduced. The EFS was originally developed as a primitive file storage capability for the GCE mass storage devices.

The two principal design objectives of the EFS are:

1. Sufficient functional capability to support the Cronus distributed file system.

2. Simplicity and efficiency.

   The principal users of the EFS will be object managers. Client processes will seldom, if ever, directly access files through the EFS. Therefore, only the most basic file operations need be supported. More complex file functions can be supported by the object managers themselves. Simple steps have been taken in the internal organization of the EFS to support effective crash recovery and system restart procedures.

The Elementary File System will have the following characteristics:

1. The name space for EFS files is flat. Names for EFS files are called FileIDs, and they are fixed length bit strings. FileIDs are not Cronus UIDs. A FileID is unique on the EFS that generated it, but it is not unique across all Cronus hosts. The EFS is a Cronus object in much the same way that the existing UNIX or VMS file systems are Cronus objects, but

2. A EFS file is not a Cronus object.

3. File data is organized as a sequence of fixed length blocks. File i/o is sequential, and is block oriented. The basic file i/o operations are.

   ReadEFSFileBlock(FileID, BlockNumber, Buffer), and
   WriteEFSFileBlock(FileID, BlockNumber, Buffer).

4. There are no open or close operations. No setup is necessary to read data from or write data to an existing EFS file.

5. It is necessary to create a EFS file before writing data to it. This is accomplished by the

   CreateEFSFile()

   operation, which creates an empty EFS file and returns its FileID.

6. EFS files are deleted by the

   DeleteEFSFile(FileID)

   operation

7. There is no access control for EFS files. Possession of the FileID for a EFS file is sufficient to access the file.

The EFS will normally be accessible only to Cronus Services. The primal file manager is an example of such a service. These services provide controlled access to the objects and operations that they implement, as described in Section 8.

In addition to supporting the local primal file manager, the EFS may be operated on as an object to permit remote access for maintenance and debugging purposes. There is a single access control list (ACL) associated with access to the entire EFS through the EFS_File Manager. Only a very few principals will be on the ACL for a EFS. An example of a principal which might be granted access to the EFS is the "System Maintenance" principal.

8.4.2 File Formats

The following description of the Elementary File System structure assumes that a disk can be represented by a series of fixed length blocks. In the Cronus ADM, the storage may be:

a disk drive on a GCE,

a disk device in a UNIX system, or

a contiguous file on the VAX/VMS.

The EFS makes few demands on the underlying recording medium, and it is relatively easy to see that most potential Consituent Operating Systems will provide a construct upon which the EFS can be built.

File disk blocks are self-identifying for reliability purposes. Each block includes a header that contains the FileID and the block number. The file header in each block contains a NextBlock pointer which is the disk address of the next block, if any, in the file. The NextBlock pointer in the last block contains a special value marking the end of file.

There is a FileID Table which provides a mapping between FileIDs and the disk address of block 0 of the file (see Figure 1). The FileID Table is as a file with a well-known FileID (FileID = 1). Its block 0 will be stored at a known disk address (with an alternate copy stored at another location to prevent loss of data in case the primary block is bad) The FileID Table is a hash table.

There is a FreeDiskBlock table which records the disk blocks that are available. The FreeDiskBlock table is a bit table stored in a file with a well-known FileID (FileID = 2). Its block 0 is stored at a known disk address. When a file is deleted, its blocks are recorded in the FreeDiskBlock table, and the FileID field in the headers of each of the blocks is cleared. As disk blocks are needed they are allocated using the FreeDiskBlock table.

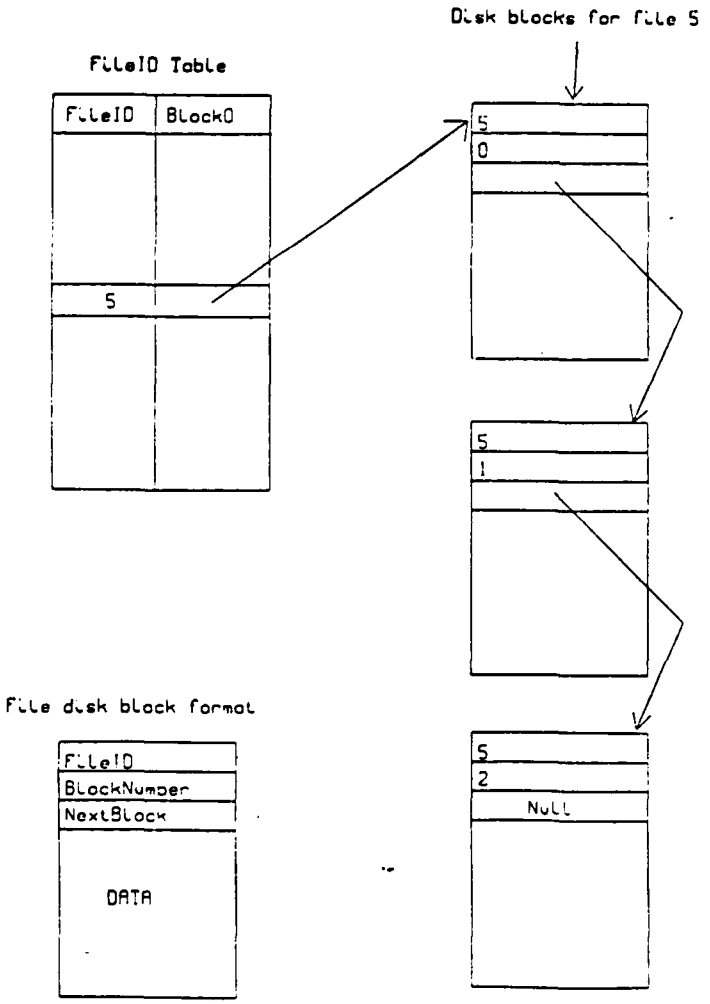
There are two types of EFS files. The type of the file is contained in the header of block 0. T types of EFS files are (see Figure 2).

a. Short file.

This is a file, all of whose data will fit within block 0.

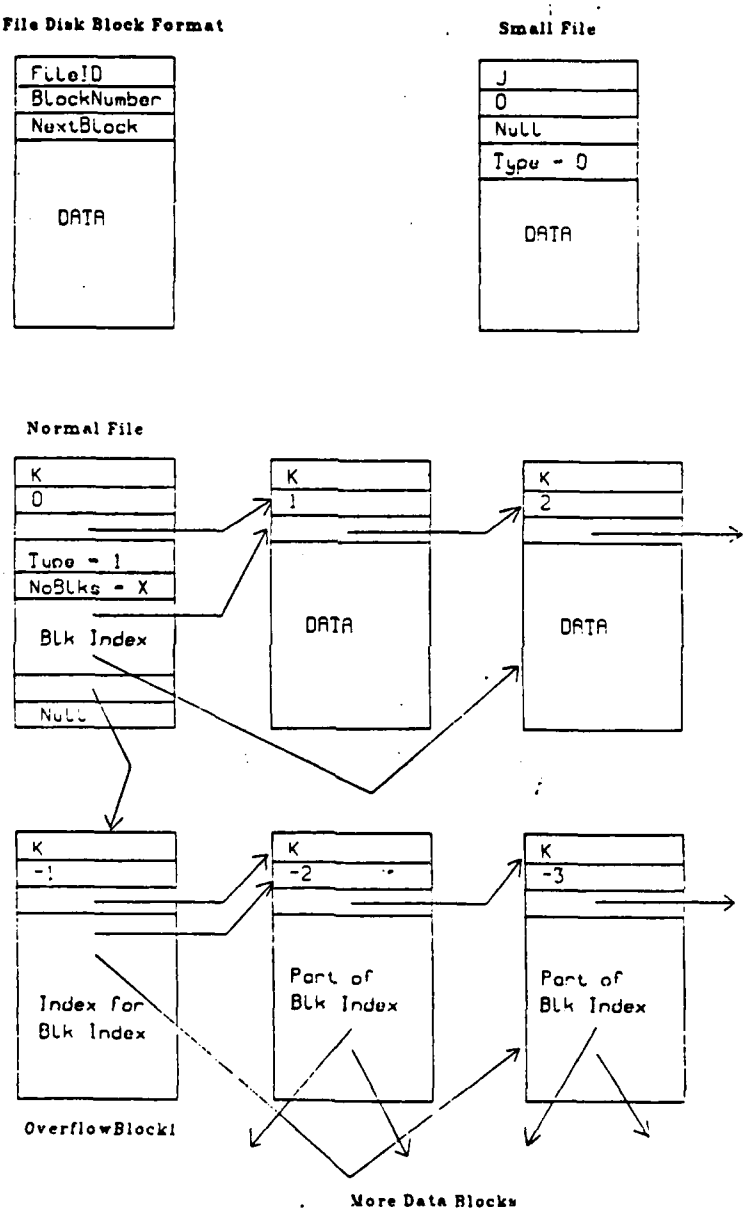
b. Normal file.

This is a file whose data will not fit within a single block.



EFS File Table
Figure 8.1

Random Access GCE Files



EFS File Types
Figure 8.2

A Normal file may contain index blocks which allow random access to the file. By convention, the first of these blocks is given block number -1, and contains:

i. A block index which holds the disk address of blocks 1 through N of the file; and
ii. The disk addresses for two overflow blocks, named OverflowBlock1 and OverflowBlock2, which can be used to find the block index entries for blocks numbered greater than N.

If the file is very large, not all of its index will fit into block -1.

OverflowBlock1 is used as an index for blocks which store part of the block index which will not fit in block -1. Specifically, if block -1 can store indices for blocks 1 through N, if OverflowBlock1 can store M disk addresses as indices, and if each block it indexes can store P disk addresses, OverflowBlock1 can provide access to indices for M*P additional blocks, numbered (N+1) through (N+M*P). The block index for the Normal file shown in Figure 2 overflows block -1 into OverflowBlock1, and is small enough that it doesn't require OverflowBlock2

OverflowBlock2 provides an additional level of indirection for very large files. It contains an index for blocks which are used in the same manner OverflowBlock1 is. If OverflowBlock2 can hold Q disk addresses as indices, then it can provide access to indices for M*P*Q blocks, numbered (N+M*P+1) through (N+M*P+1+M*P*Q).

By convention the BlockNumber for OverflowBlock1 is -2. Any index blocks referenced by OverflowBlock1, as well as OverflowBlock2 (if present), and any index blocks it references directly or indirectly are assigned BlockNumbers in a negative sequential fashion starting at -3 in the obvious manner.

Some constituent hosts will have multiple disks (in the case of UNIX, these may actually be disjoint regions on a single physical disk, and in the case of VMS, they would be multiple contiguous files). Part of the FileID specifies the disk on which the file resides. The CreateEFSFile operation takes an optional parameter which specifies a disk. If the parameter is supplied, block 0 and all subsequently created blocks of the file

are allocated on the specified disk. If the parameter is not supplied, block 0 and subsequent blocks are allocated on the disk the EFS sees fit.

8.4.3 Disk Salvaging

There is a BadDiskBlock table which holds the disk addresses of bad disk blocks. The BadDiskBlock table is stored in a file with a well-known FileID (FileID = 3).

There is a EFS disk salvage operation which can reconstruct the FileID table, the FreeDiskBlock file, and the BadDiskBlock file, and reset the NextBlock pointers in files.

The salvager may encounter files with missing blocks. When it does, it will fill in any hole it encounters with a newly allocated filler block, linking the filler block into the file where the hole was. The FileID of the filler block will be set to the ID of the file, and its BlockNumber will be set to a special BlockNumber which identifies it as a filler block. The only data in a filler block will be the BlockNumbers of the previous and next file blocks which contain data. Higher level software can be used to recover the data in a file which contains filler blocks.

As the salvage procedure encounters bad disk blocks, it records them in the BadDiskBlock file. If it encounters a bad block which is part of a file, the salvager will remove the block from the file and substitute a newly allocated replacement block by linking it with the other blocks of the file in place of the bad block. The FileID of the replacement block will be set to the ID of the file, and its BlockNumber will be set to a special BlockNumber which identifies it is a replacement block. The only data in the replacement block will be the BlockNumber of the block it replaces. This will make it possible for higher level software to recover the data in other blocks of the file.

9 Symbolic Naming

9.1 The Cronus Symbolic Name Space

Cronus has a global symbolic name space with the following properties:

1. Cronus symbolic names are location independent.

   a. A name for an object is independent of its host.

   b. A name that refers to an object can be used regardless of the location from which it is used.

2. Cronus symbolic names are uniform.

   Common syntactic conventions apply to names for different types of objects.

The symbolic name space is constructed upon a hierarchically structured tree. The tree contains nodes and directed labeled arcs. There is a distinguished node called the "root" 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. Links provide an overlaid structure based on symbolic pointers which provide a name space which is a network, so a node may be reached by more than one path.

Non-terminal nodes (those from which arcs may originate) are called directories. Each labeled arc corresponds to a catalog entry. The label for an arc is called an "entry name".

The complete name of a node, which is the symbolic name for the object, is formed by concatenating the labels on the arcs traversed on the path from the root node to the node in question, separated with the character ":". In other words, 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 "} s {*" means zero or more occurences of s.

It is also possible to name nodes relative to a directory. Such a relative 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 relative name is:

```
} x . {* y
```

There are conventional names for the current ("connected" or "working") directory, its parent, and the user's initial directory.

The most common types of cataloged objects are the various kinds of files, but any other object may be cataloged. Some conventions will be adopted; for example, there will be a :dev directory which contains the symbolic names for the devices on the system. These conventions are not enforced by the system, and any object may be entered into any directory (assuming appropriate authorizations) at the convenience of the user.

There are certain special object types which are used in support of the catalog itself, including:

- o Directories

  A directory object (type CT_Directory) is a non-terminal node in the catalog tree.

- o Links

  The catalog entry for a link (type CT_Symbolic_Link) identifies another point in the symbolic name space called the link target. These objects are stored in the catalog itself. Links are cataloged as terminal nodes in the name hierarchy tree. Links are handled specially within the Lookup operation.

- o External linkages

  An external linkage (type CT_External_Linkage) is an object which implements access to another name space. External linkages are cataloged as terminal nodes in the name

hierarchy tree. External linkages permit users to refer to non-Cronus objects directly from the Cronus name space. For example, an external linkage might be used to give a file directory on a Cronus application host a Cronus symbolic name.

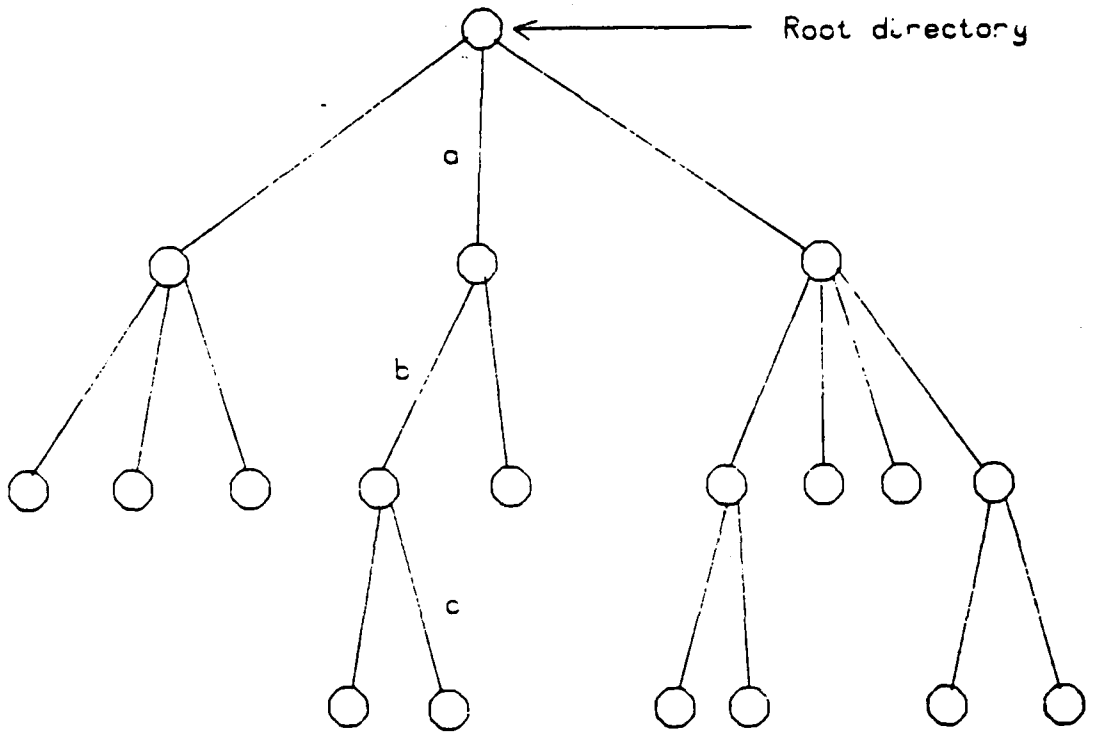
For some object types it is useful to be able to think of a collection of the objects as a sequence of "versions" or "revisions" of the same logical object. The Cronus Catalog implements a version feature for certain types of objects; for example, versioning will be supported for files, but it will not be supported for directories.

For types for which versioning is supported, the catalog entry operation will permit the same name to be entered into a directory more than once. Each copy of the entry will have a distinct version field and should point to a different object. However, all objects pointed to by different versions of the same entry name must be of the same type. The first time a name is entered, the result will be version 1 of the object. Subsequent entries of the same entry name will result in successively higher versions of the object. All of the catalog operations which take a name parameter will allow the specification of a version number as well.

The catalog managers provide routines that can scan through the catalog and return catalog entries for names that match a specified pattern.

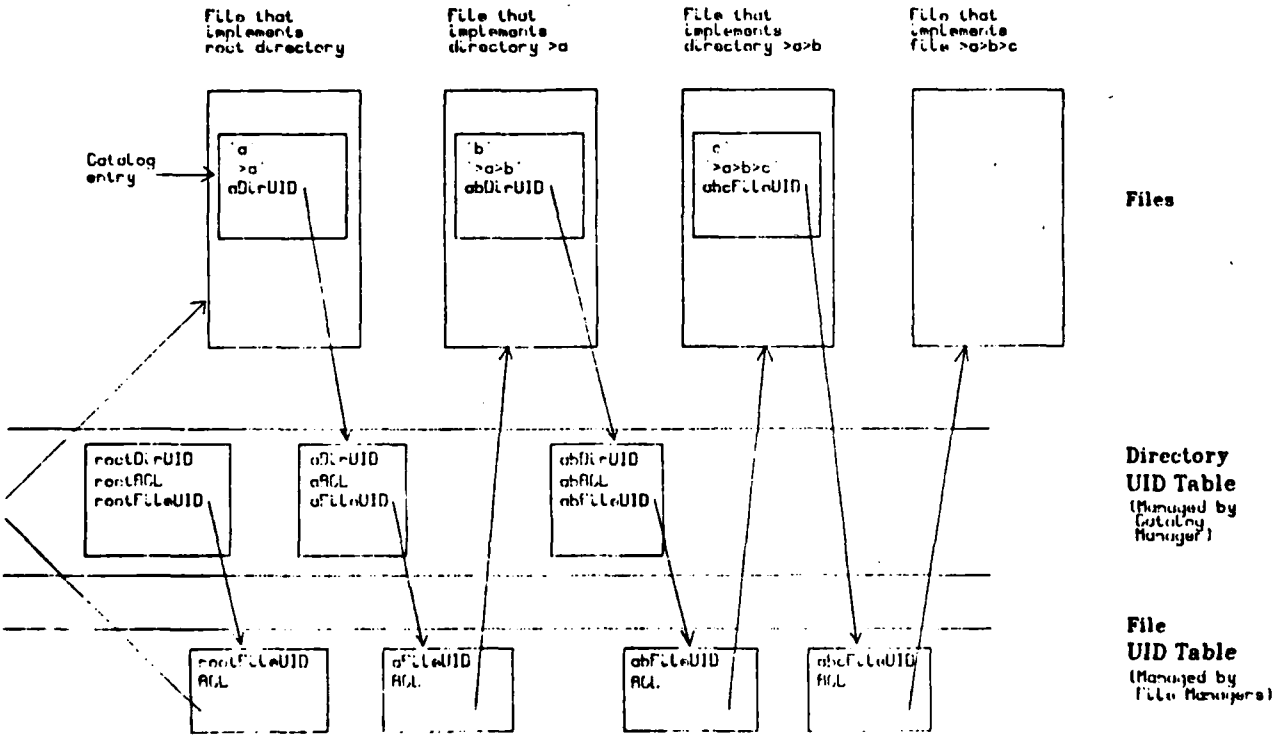
The create(entry) operation can be used to simply establish a symbolic name for a Cronus object of any type except a directory, symbolic link, or external linkage object. These types of entries are inserted in the catalog when they are created (since other objects need not be named, the creation of the object and naming of the object are distinct operations). In a sense, these objects are special in that they must have a symbolic name in addition to a UID.

Figure 1 shows a relatively simple symbolic name tree and Figure 2 shows part of the underlying directory structure that corresponds to the part of the tree that contains the name .a.b.c.



Catalog Hierarchy
Figure 9.1

Implementation of Cronus Catalog



Implementation of Cronus Catalog
Figure 9.2

When a lookup operation is invoked, the catalog manager interprets a complete Cronus symbolic name by starting at the root directory. The UID of the root directory is well-known. The catalog manager processes a name component by searching the current directory for a matching catalog entry. If it finds a matching entry and there are no more name components, the lookup is complete and it returns the catalog entry. If it finds a matching entry and there are more name components to interpret, the entry must be for a directory, symbolic link, or external linkage, or else the lookup ends in failure. If the entry is a directory, the catalog manager continues the lookup by obtaining the UID for the directory from the entry and then using it to interpret the next component. Interpretation of a relative symbolic name is handled in the same fashion, differing only in where the lookup starts. For a relative name, the catalog manager starts its search at the starting directory parameter of the lookup operation.

Symbolic links encountered during lookup are handled in a special manner. When a link is encountered, a new name is formed by substituting the link target, which is a complete Cronus symbolic name held in the catalog entry, for the portion of the symbolic name evaluated so far. The lookup operation then resumes by interpreting this new name. Links can be thought of as macros which are expanded during the lookup operation.

A parameter of the lookup operation controls whether links are to be expanded. If the parameter specifies that links are to be expanded, the substitution of link targets during the lookup operation occurs. If the parameter is set to prevent links from being expanded, the lookup operation terminates when a link is encountered. In this case, the lookup operation will be considered successful if the name has been completely evaluated. Otherwise it will be considered a failure.

9.2 Objects Related to the Catalog

9.2.1 Objects of Type CT_Catalog_Entry

Each catalog entry is a Cronus object; however, unlike most objects in Cronus, a catalog entry has no UID. A catalog entry contains the following information:

UID for the object,
Complete symbolic name for the object,
UID for creator of entry (PrincipalUID); and
Type-dependent information.

Type-dependent information for objects of type CT_Directory, CT_Symbolic_Link, and CT_External_Linkage is discussed below. For objects that are not part of the Cronus catalog, everything that can be known about an object is maintained by (or can be obtained from) the manager for the object. That is, no type-dependent information is maintained in the catalog.

9.2.2 Objects of Type CT_Directory

For directories, no type-dependent information, except the host that stores the directory, would be maintained in the catalog entry. All other information about the directory will be maintained with the directory object itself.

9.2.3 Objects of Type CT_Symbolic_Link

For a symbolic link, the type-dependent information, which completely specifies the link, consists of the complete symbolic name for the link target.

UID,
Complete symbolic name for the link,
UID for creator of entry (PrincipalUID), and
Complete symbolic name for the link target.

9.2.4 Objects of Type CT_External_Linkage

For an external linkage, the type-dependent information completely specifies the external linkage. It includes a Cronus interpretable designator for locating the other name space and a symbolic name that is interpretable in that other name space. The details of the method for designating other name spaces and for interacting with them are incomplete. A catalog entry for an external linkage will include.

UID.
Complete (Cronus) symbolic name for the external linkage;
UID for creator of entry (PrincipalUID)
Cronus interpretable designator for the other name space; and
Symbolic name interpretable in the other name space.

9.3 Catalog Operations

9.3.1 Objects of Type CT_Catalog_Entry

The following operations are defined for the Cronus symbolic catalog (see Cronus User's Manual cat_entry(3)):

Create
Remove
Lookup
Read
Change
InitScan
ScanDirectory
LookupWild

LookupWild performs a catalog lookup using Cronus wild card conventions (see Cronus User's Manual sym_name(4)), and returns a list of all the entries which match the specification. InitScan and ScanDirectory perform the same function, but incrementally, returning individual entries.

9.3.2 Objects of Type CT_Directory

The following special operations are defined for objects of type CT_Directory (see Cronus User's Manual directory(3)):

Create
Remove

9.3.3 Objects of Type CT_Symbolic_Link

The following special operation is defined for objects of type CT_Symbolic_Link (see Cronus User's Manual sym_link(3)):

Create

9.3.4 Objects of Type CT_External_Linkage

The following special operation is defined for objects of type CT_External_Linkage (see Cronus User's Manual ext_link(3)):

Create

9.3.5 Access Control for Catalog Operations

All of the catalog operations are operations on one or more directories. There are three rights defined for access control purposes:

ReadDirectory,
WriteDirectory, and
ModifyACL.

ReadDirectory rights are needed for all operations which return information from a directory. In operations which access multiple directories, such as Lookup, ReadDirectory rights are needed for each directory accessed. WriteDirectory rights are needed for all operations which insert or remove entries from a

directory, or alter the contents of an entry (with the exception of those which change the access rights). ModifyACL rights are needed in order to change the access rights to an object represented by a catalog entry.

Table 9.1 summarizes the access rights required for the various operations.

	Read Directory	Write Directory	Modify ACL
Create(entry)		x	
Create(link)		x	
Create(external linkage)		x	
Remove(entry)		x	
Lookup	x		
LookupWild	x		
InitScan	x		
ScanDirectory	x		
Read(entry)	x		
Change(entry)			x
Create(directory)		x	
Remove(directory)		x	

Table 9.1 Access Rights Required for Catalog Operations

9.4 Catalog Implementation

9.4.1 Introduction

The following implementation issues are discussed below:

1. the manner in which client processes interact with the catalog manager which implement the catalog functions.

2. the use of Cronus data storage resources to implement the catalog data base.

3. the distribution of the catalog data base among Cronus hosts;

9.4.2 Cronus Catalog Managers

There is a catalog manager process at each host that maintains part of the catalog. It is the object manager for objects of types CT_Cronus_Catalog, CT_Catalog_Entry, CT_Directory, CT_Symbolic_Link, and CT_External_Linkage.

The catalog managers communicate with client processes by means of the standard Cronus IPC facility. Since the catalog hierarchy is distributed among Cronus hosts, different managers will have direct access to different parts of the catalog. Some catalog operations can be accomplished by a single catalog manager and some require the cooperation of two or more catalog managers.

For example, the Remove(DirUID, catEntUID) operation would normally be sent to the manager for directory DirUID, and only that manager is required. The lookup operation may require catalog managers on two hosts if the manager to which it is sent does not contain the subtree required to interpret the entire symbolic name.

A client process will not, in general, know which catalog manager is the best one to perform a given operation. For this reason, a client can initiate a catalog operation with any catalog manager. If the manager selected can perform the operation requested by itself, it will. If not, it will interact with other managers as necessary to perform the operation.

9.4.3 Implementation of the Catalog Hierarchy

Directories are stored in files. The catalog manager maintains a UID table for the objects it manages. Since the principal objects implemented by the catalog manager are directories, this table is called the Directory UID Table. The

Directory UID Table maps the UIDs for directories into their object descriptors.

A directory contains zero or more catalog entries. The catalog entry for a (inferior) directory contains the UID of that directory. To access a directory given its UID, the catalog manager uses the Directory UID Table to obtain the object descriptor for the directory, and then uses the file UID in the descriptor to access the file that holds the directory.

9.4.4 Distribution of the Catalog

9.4.4.1 Principles Affecting Distribution

Among the considerations influencing catalog distribution are.

1. The catalog should not be stored at only one site.

   This is a reliability consideration.

   The catalog should be distributed, and it should probably be replicated in some fashion.

2. The entire catalog should not be stored at any single site.

   This is a scalability consideration.

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

   This is a reliability consideration.

   Access to objects through the UID name space has this property since the information required to access an object, given its UID, is maintained by object managers. Access to objects through the symbolic name space should also exhibit it.

The catalog entry for an object (or a copy of the entry) should be stored at the same site as the object. In addition, there should be enough information at the object site to control access to the object.

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

   This is a common sense consideration.

   It is not necessary to replicate catalog entries for objects beyond that required by (3).

There are some further issues to consider associated with (2) and (4), and we discuss them in more detail in the next two subsections. The discussion includes elements of the implementation of the reliable system as well as the primal system, because these may impose constraints on the primal system design.

9.4.4.2 Dispersal Of The Catalog

This section examines the requirement that the catalog not be stored at a single site. The line of reasoning followed is essentially that that lead to the design of the Elan hierarchy [BBN 3796].

Directories are the basic unit of distribution for the Cronus catalog. Directories are implemented by Cronus primal and reliable files. The lookup operation follows the components of a symbolic name through a number of different directories, one for each component in the name (assuming it does not encounter a symbolic link). Unless there is a further restriction on the dispersal of the catalog, each directory could be at a different site from the previous one.

It is desirable to limit the number of sites that must be visited in a lookup operation. Two useful restrictions are 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.

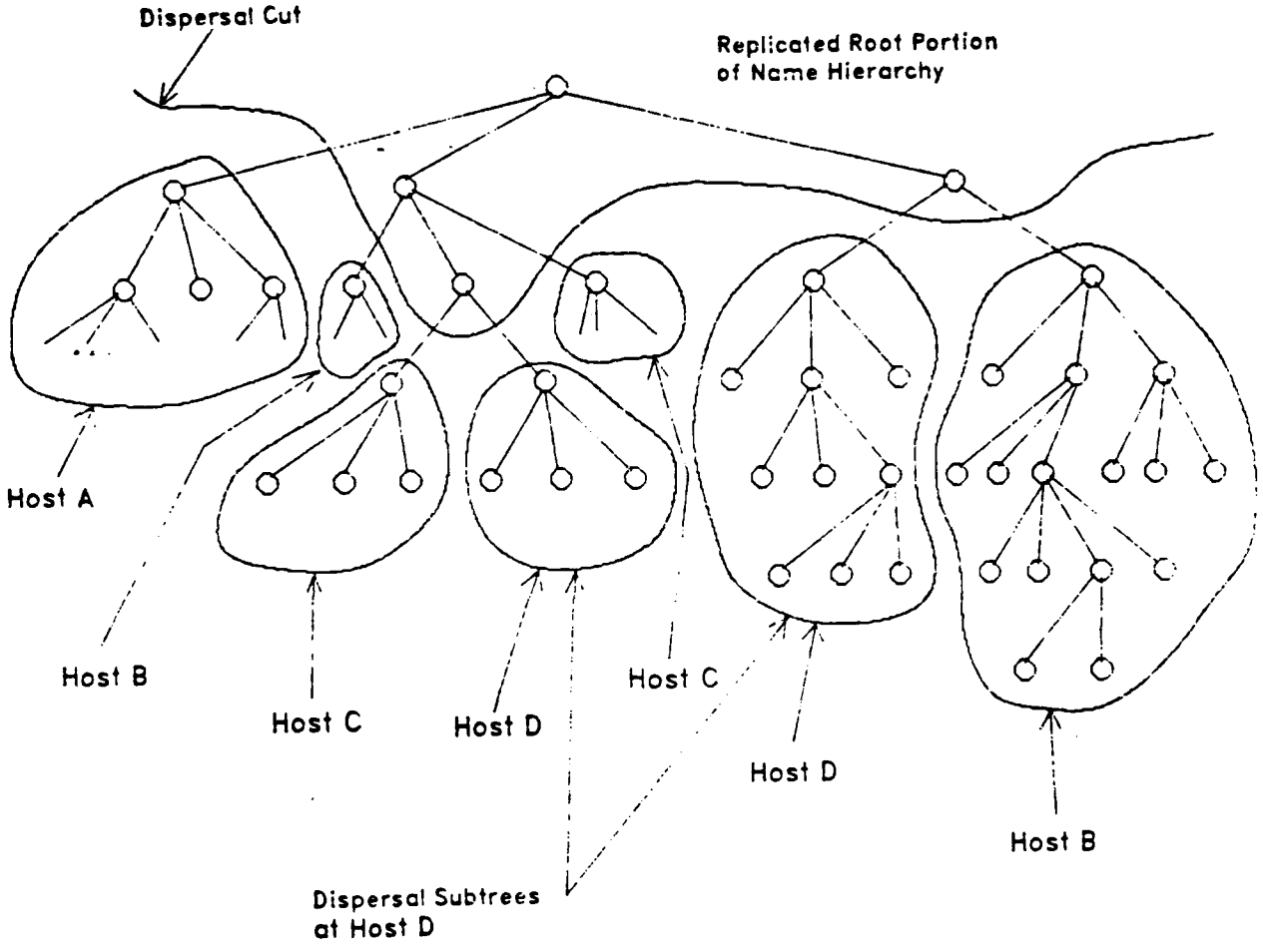
Restriction 1 ensures that lookup operations within a subtree that is below the dispersal cut can be confined to a single site. Restriction 2 ensures that the task of determining the site that stores a particular dispersal subtree can be confined to the site that stores the root portion of the hierarchy. As a result, lookup operations require at most two catalog sites.

It is useful to add a third property to the dispersal of the catalog.

3. The root portion of the catalog hierarchy should be replicated. Furthermore, a good way to replicate it is to maintain it at each site that maintains a part of the catalog (i.e. a dispersal subtree). The reasons for doing this are:

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

Figure 3 illustrates how a simple name hierarchy might be dispersed among several hosts according to these three restrictions.



Dispersal of the Catalog
Figure 9.3

For this to be practical, it must be possible to maintain the copies of the root portion in a consistent fashion among the same set of hosts that store parts of the catalog. It has been observed that the root changes very slowly, because few users are authorized to make changes, and because changes generally occur as the result of the addition or deletion of a user or project. This means that the maintenance mechanism need not be powerful enough to handle the general multiple copy update problem.

9.4.4.3 Replication of Catalog Information

The primary consideration for replicating catalog information is one of reliability. The objective is to ensure that Cronus objects with symbolic names are accessible symbolically whenever the sites that manage the objects are. It is likely that unavailability of a catalog manager will be the result of a host crash, so that we can assure maximum access by providing a copy of a catalog entry on the host where the object is cataloged. Then the entry will usually be available whenever the object is. If this is the same as the site of the primary catalog entry, then no replication is needed. If it is different, then a secondary catalog entry is provided on the host where the object resides.

For every host on which there are object managers, there will be either a full catalog manager or a secondary catalog manager. Each full catalog manager will maintain a fully replicated root part of the catalog tree and its own subtrees rooted at the dispersal cut. In addition, both full and secondary catalog managers will maintain a separate database, the secondary entry table, which contains secondary catalog entries for objects which are on its host but for which the catalog subtree containing it is not local.

A secondary entry is a catalog entry which stands in for the primary entry for an object. It differs from the primary entry in two respects. First, it can reside only on the host on which the object resides, and then only if the primary entry is on a different host. Second, it is stored in the secondary entry table, not in a directory.

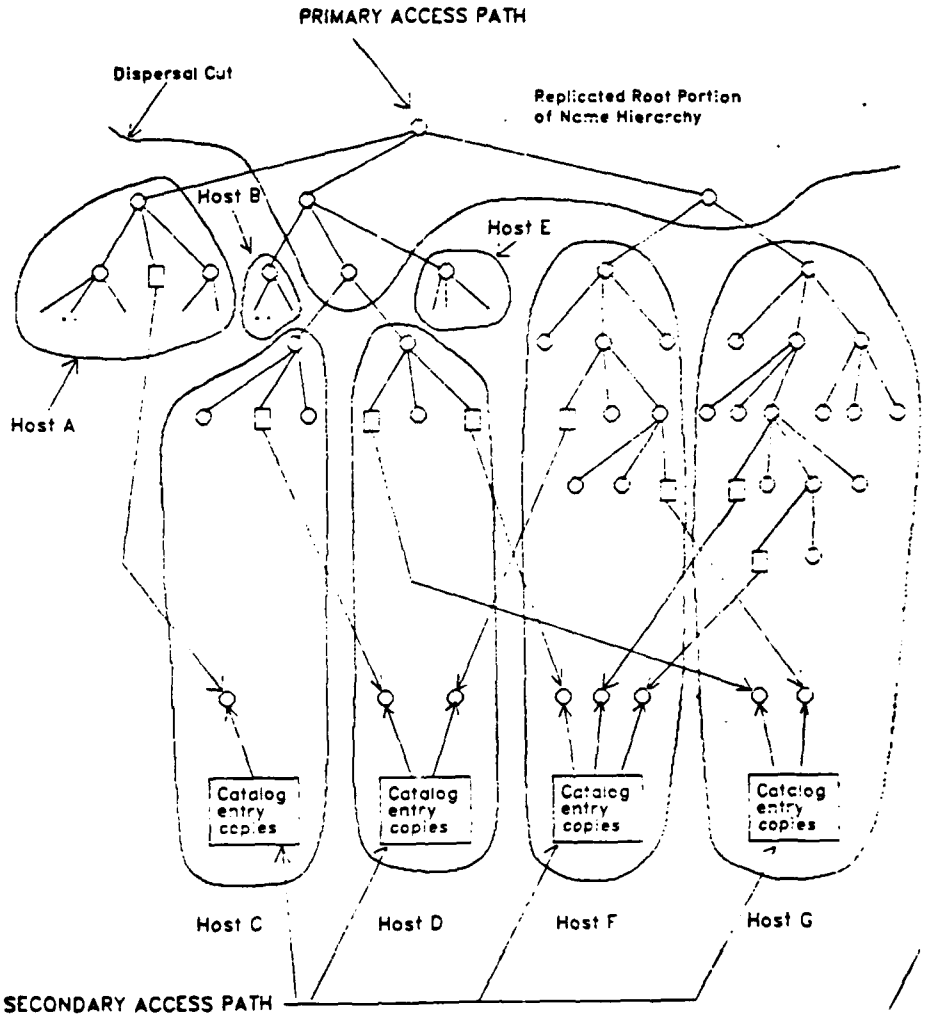
The secondary entry table is not used to speed up local access to the object UID. Rather, it is available only to support catalog operations when the primary entry is not accessible. The reason for restricting its use is to avoid synchronization problems between the primary and secondary entries for an object during normal operation of the system when no hosts are down. If the object has more than one symbolic name, a copy of each catalog entry will be stored at the object's host. That is, there will be a collection of Cronus catalog entries at each host for those objects that have symbolic names that require access to directories on other hosts. The catalog manager software will maintain the consistency between these secondary catalog entries and the primary entry.

Figure 9.4 illustrates how the catalog information will be maintained. The circular nodes represent objects that are stored at the same host as their entry in the catalog hierarchy and the square nodes are used to represent catalog entries for objects that are stored remotely from their entries.

Under normal conditions, the lookup operation uses the primary entries in the symbolic catalog. When not all of the directories are accessible, the secondary symbolic access path is used. The lookup will succeed whenever the object itself can be reached, since if the object has a symbolic name, a copy of the catalog entry object will be stored at the site that manages the object. (14)

When a client process first invokes a Lookup operation, the operation is performed using only the primary catalog entries. If that fails, the client may then attempt to perform a look up on the full symbolic name of the entry of interest. In this second lookup attempt, the client must multicast the lookup request to all catalog managers and set the key in the request

(14) Lookups of partial symbolic names cannot be performed using the secondary access path, because the failure of the initial lookup suggests that the catalog manager which can interpret the Directory UID for the start of the search is unavailable. To use the secondary access path, the client must remember the full symbolic name for entry. Further, the secondary access path will not have the mechanism of symbolic links available. As a result, a path name utilizing such a link will also fail.



Secondary Symbolic Access Path
Figure 9.4

indicating a secondary access path lookup.

Lookup by means of the primary path is much more efficient since it is directed, whereas lookup by means of the secondary path is undirected. There is no a priori knowledge of the host or hosts that need to be consulted to perform a lookup by the secondary path.

9.4.4.4 Synchronization Among Catalog Managers

There are two cases in which catalog managers must synchronize among themselves in order to preserve consistent information: the replication of the catalog hierarchy above the dispersal cut, and the correspondence between the primary and secondary entries for objects which reside on one host and have their primary catalog entry on another.

In addition, there are two aspects of the synchronization problem, the first is the synchronization among hosts which are all running, the second between a host which has changed the catalog and a host which is reintegrating into the Cronus cluster after a period of inaccessibility.

This section discusses techniques for automating replication of the root portion of the Cronus hierarchy (i.e. that above the dispersal cut). While the approach discussed applies to the Cronus catalog, it is also intended to be used as a base for more general replication services that might be applied to other Cronus components (the authentication manager, file managers, etc.)

As with all Cronus functions, automation of catalog replication will be implemented by the object managers. Initially, we can think of the functions needed to implement this automation as being composed of the basic operations on an object type. Later it might be appropriate to cast them as new operations. In any case, we will refer to these functions as operations irrespective of their actual implementation as operations on a type. In the case of the catalog, these replication functions will be handled by the Catalog Managers, rather than in a more general way such as through some form of replicated file. Eventually, when we

gain more experience with replication, we may want to provide a more generic way of providing replication services.

We define the following basic operations:

- o Replicate existing directory
- o Dereplicate existing directory
- o Modify existing directory (add, delete, modify/entry)
- o Reintegrate host

In addition to these operations, we can add two more functions related to management of the replicated portion of the catalog:

- o Move dispersal cut (or replicate/dereplicate above/below directory)
- o Copy dispersal (make a copy of the entire dispersal hierarchy)

In order to simplify the design, we will restrict ourselves to these functions. Other variants, such as create a new replicated directory, can be implemented from these and the existing catalog operations in the obvious manner.

Our approach to maintaining consistency in the replicated portion of the hierarchy will be to use update logs that are maintained and accessed by the Catalog Managers. We will discuss the management of updates in more detail later, when we discuss reintegration.

Before discussing the operations, recall that all directories in the hierarchy above the dispersal cut are replicated on all hosts. Below the dispersal cut, each subtree of the catalog hierarchy is maintained on a single host. This ensures high availability of the root portion of the catalog and a minimal number of inter-host accesses in a directory search. The catalog is designed to accommodate infrequent changes to the root portion of the hierarchy, so speed of update is not a major issue.

In Figure 5 we see a detailed representation of the replication of the root portion of the catalog hierarchy on two hosts, A and B. Note that the directories above the dispersal cut are truly replicated, having the same directory UIDs. The reader should not confuse the implementation of

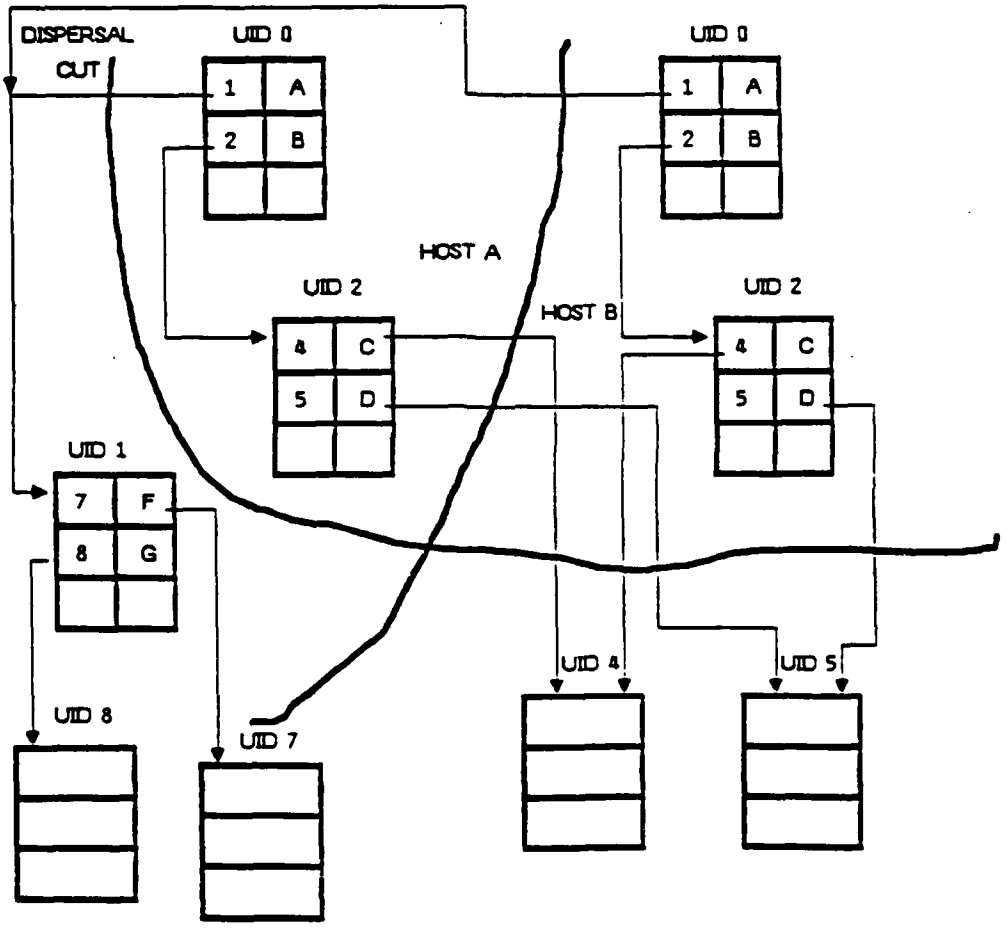
the catalog (as files with different file UIDs maintained by the file managers) from the replication of the catalog itself (directories with the same UIDs maintained by catalog managers). The reader should also remember that the contents of the replicated directories are also replicated (e.g. they have the same entries), and that they have location independent semantics. That is, the entries consist of a symbolic name that is known globally (through the catalog) and a UID that is known globally (through the operation switch). With this background, we can now go on to discuss the operations in more detail.

9.4.4.5 Replicate

The replicate function takes a specified non-replicated directory and replicates it throughout the configuration. That is, a copy of the directory and all its entries will be created by the Catalog Manager on each host in the configuration with the same UID as the source. The function is restricted in that only the root directory or children of replicated directories can be replicated. To ensure consistency, a copy of the newly replicated directory is made available available by the Catalog Manager when the operation has been completed locally. Thus, only when the new directory is allocated and its entries are copied is it made visible by inserting its UID into the Catalog Manager's UID table. Each copy of the directory is also marked as being replicated to assist the Catalog Manager in its future management. The operation is managed by the Catalog Manager of the source directory which communicates directly with all the other Catalog Managers in the configuration to complete the operations on their hosts.

The replicate operation is logged by the initiating Catalog Manager to allow reintegration of hosts which cannot complete the replication immediately. We will discuss update log maintenance and reintegration later. For now, we note that a log entry is created for the operation and hosts that have not completed it will use the log in the process of reintegration at a later time.

Replication in the Cronus Catalog



Replication in the Cronus Catalog
Figure 9.5

The following pseudo-code describes the algorithm:

```
REPLICATE DIR
   IF DIR ALREADY REPLICATED OR PARENT NOT REPLICATED THEN
        ERROR

   ELSE
        LOG REPLICATE DIR
        MARK LOCAL DIR REPLICATED
        FOR ALL REMOTE CATALOG MANAGERS

              CREATE REPLICATED COPY OF DIR
                   /* CREATE, COPY ENTRIES, MARK DUPLICATED,
                        MAKE VISIBLE*/
```

There are several issues that are raised by this method aside from those of log file management. First, the algorithm requires that there be a database of all hosts in a configuration that run the replication service. The database should be distributed on all hosts for efficiency and availability.

The second issue is whether the remote replications should be managed synchronously (waiting for remote operation to complete) or asynchronously (telling the remote Catalog Manager to start the operation and not waiting for completion). If the operation is synchronous, there are obvious performance implications for completion depending on how long the operation will take. For a large configuration this could be a problem. A time-out will be required for those hosts that are down or cannot respond. Asynchronous management means that it is hard for the originator to know when and if the operation was completed. It puts more of a burden on the reintegration procedure for making sure the operation is carried out successfully. One possibility in the asynchronous case is for the target to acknowledge start of the operation and not have the originator wait for completion.

The issue here is the definition of when an operation is complete. Strictly, an operation is complete only when all hosts in the configuration have successfully completed it. However, it may be sufficient to consider an operation "complete" from the point of view of the initiator when it has been successfully logged and all running hosts in the

configuration have been notified to start the operation. Since the reintegration procedure will presumably eventually cause the operation to be completed on all hosts, relying on it to make sure the operation is completed on all hosts is probably adequate. Thus, the initiator's responsibility is to a) log the operation; b) notify all running hosts in the configuration to start the operation; and c) complete the operation on the local host. Once the operation is successfully logged, we assume that it will be completed on all hosts eventually by the reintegration procedure even if any of the hosts (including the initiator) crashes in the midst of an operation

The only problem with this approach is if a host cannot complete the operation operation due to problems such as lack of resources (e.g., no space to add new directories, etc.) In these cases, the best solution is probably to notify the operator of the resulting inconsistency through error logging or the monitoring and control system so that the problem can be manually resolved. The reintegration procedure can still be used in these cases to retry the operation at a later time, but presumably operator instruction will be required in some instances to clear up the cause of the problem.

Another issue in the design of replication functions is maintenance of the secondary catalog database. Recall that to maintain accessability of symbolically named objects, a secondary catalog entry is maintained on the host where an object resides if the object is located on a different host than its primary catalog entry. Thus, objects will be accessible symbolically through this secondary path even if the primary path is unavailable. However, in the case of the replicated portion of the catalog, the need for the secondary database is obviated since the catalog information is already available on all hosts in the configuration.

9.4.4.6 Dereplicate

The dereplicate function takes a specified replicated directory and removes all copies of it except the one on the host of the originator (which can be any host in the configuration). Dereplicate only applies to replicated directories whose children are not replicated. The algorithm is similar to replicate in that it takes place available: first the directory is made invisible on the remote host, then the remote copy is removed. The following pseudo-code summarizes the operation.

```
DEREPLICATE DIR
     IF DIR NOT REPLICATED OR ANY CHILDREN REPLICATED THEN
          ERROR

     ELSE
          LOG DEREPLICATED DIR
          MARK LOCAL DIR DEREPLICATED
          FOR ALL REMOTE CATALOG MANAGERS

                DEREPLICATE DIR /* MAKE INVISIBLE, DELETE DIR */
```

One issue with dereplicate is how to preserve the characteristic that subtrees of directories below the dispersal cut be contained on a single host. One solution would be to force this condition to be true before the directory could be moved below the dispersal cut (dereplicated). This would require manual reorganization of the directory hierarchy before dereplication. Another approach might be to relax this constraint and allow the dereplication to take place anyway. As an optimization, the hierarchy could be reorganized manually later to meet the condition.

9.4.4.7 Modify

The modify replicated directory operations (add, delete, change) also proceed along the lines of replicate/dereplicate, adding the operation to a log file and notifying all the remote Catalog Managers to complete the operation.

Modification of the existing directories presents a more severe synchronization and locking problem than replication and dereplication. For replication and dereplication, atomicity of the operations ensures consistency, since the directory will be available somewhere, by virtue of the Cronus IPC system (i.e. UIDs are location independent) even if it is not yet fully replicated. Modification, on the other hand, could lead to inconsistency if the operation is not completed successfully or if simultaneous modifications to the same directory are attempted.

Clearly, some form of concurrency control is needed to prevent conflicts and inconsistencies. Because changes to the root portion of the hierarchy occur infrequently, we can prevent conflicts (simultaneous changes to the same entry) by locking the root portion when any change is made, so that only one change can occur at any time. Since modification of the root hierarchy is an administrative function, this is probably acceptable.

Inconsistency in the root portion of the hierarchy is a different problem which results from latency in completing the operation across all copies of the hierarchy. This results in periods where the directories have different contents. This may or may not be a problem in practice, depending on how frequently changes to the root portion are made.

9.4.4.8 Update

So far, we have avoided the problem of hosts that cannot complete replication operations, either because they are down during an operation or because they are isolated through network failure or partition. We have mentioned that the approach we will take for reintegration is the use of pending actions logs where each operation is recorded until completed by all hosts in the configuration. We now discuss the details of reintegration and log file management.

The basic idea is that there is some log file accessible to the Catalog Managers on all hosts. Entries in the log are made for each replication operation. A host's

catalog manager reads the log file when it comes back up and before it accepts any new requests. For each entry in the log not completed by that host, the indicated operation is completed and the host marks the operation complete in the entry. When all hosts in the configuration have completed the indicated operation, the entry is freed for garbage collection.

Entries in the log file consist of an operation code (replicate/dereplicate directory, add/delete/modify directory entry), arguments to the operation (directory UID or actual entry contents), and a vector of operation done bits corresponding to each host in the configuration. As the operation is completed by a host, its completion bit is set by the remote host's Catalog Manager at reintegration time. If all the bits are set and the entry is the last one in the log file, the file can be truncated by the Catalog Manager. Garbage collection is done by a daemon process that runs periodically to trim the log file. The assumption is that the normal state of the log file will be empty (i.e. all operations completed). In any case, as long as all the hosts in the configurations eventually come up, the log file will eventually be trimmed

Initially, at least, there will be a single central log file accessed by a global UID known to each Catalog Manager. Admittedly, this presents a weakness in the mechanism, since if the log file becomes inaccessible, updates to the hierarchy cannot be done. This can be dealt with in the future by replicating the log file on multiple hosts or by using the persistence database in each Catalog Manager to ensure the operation's completion.

The central log file can also serve as a lock on the hierarchy to serialize the updates Any access to the log file must be exclusive. This presents synchronization problems in updating either the log file or the replicated portion of the hierarchy itself. Whenever a Catalog Manager attempts a replication operation, it first tries to open the log file for exclusive access. When the initiating Catalog Manager completes the entry, it releases the log file. Again, the infrequence of most hierarchy updates should make this acceptable.

9.4.4.9 Failure Analysis

Let us look at the types of failure in maintaining the replicated database. A host can be down when the operation is started. In this case, the reintegration procedure will cause the operation to be completed when the host restarts. Another form of failure is communciation failure that isolates a host from others on the network. In this case, we assume the effects are similar to the previous case, since presumably if one host on the LAN cannot communicate with another, it is isolated from all others in the configuration. A third type of failure is inability to complete an operation because of resource limitations or some other cause unrelated to total host failure or isolation. As we mentioned earlier, the best we can do here is to report the error and wait for manual intervention to clear the source of the problem and fix the inconsistency. The latter two cases argue for running the reintegration procedure periodically, even if the host has not crashed, to restore consistency to the database in the event of a transitory failure in communications or resource limitation, etc.

A different type of failure occurs when a host crashes in the middle of an operation. Here, we want to avoid partial or incomplete results and ensure that the operation is eventually completed correctly when the host restarts. There are three mechanisms for protecting these operations from the results of such crashes. First, the initiator logs the operation as early as possible so that the reintegration procedure will be able to recover from any subsequent host crash. Similarly, hosts completing the operation do not mark the operation complete until it has actually been performed.

Second, the effects of the operation are made visible only after the result is valid to avoid partial or unusable results. Finally, enough information is available for the manager to verify whether the operation has already been completed or not, in case of a crash before the operation has been marked done in the log. This avoids the problem of a host trying to perform a completed operation multiple times. Thus when a host reads the log file it must verify that the indicated operation has not already been performed. For replicates or dereplicates of directories, it can check to see if the indicated directory already exists (or doesn't) and is

marked as replicated (or not). For addition or deletion of entries to replicated directories it can similarly search for the existence (or not) of the indicated entries. Finally, for modifications of a replicated entry, both the old and new entries must be present in the log entry, so the host can determine whether the modification was completed or not.

To summarize, the following describes the general form of the operations from the point of view of the initiator, the remote hosts, and the reintegration procedures.

```
INITIATOR.
            LOG OPERATION
            NOTIFY REMOTE CMs OF OPERATION
            COMPLETE OPERATION LOCALLY

REMOTE.     COMPLETE OPERATION LOCALLY
            MARK OPERATION DONE IN LOG

UPDATE.     LOOP. READ LOG ENTRY
                  IF OPERATION NOT MARKED DONE BY THIS HOST THEN
                       VERIFY OPERATION NOT DONE
                       IF NOT DONE THEN
                       COMPLETE OPERATION LOCALLY
                       FI
                       MARK OPERATION DONE IN LOG
                   FI
```

9.4.4.10 Other Operations

Earlier, we referred to two other functions which are important in the practical administration of the replicated root portion of the Catalog Hierarchy. The first, move dispersal cut, can be thought of as a compound replicate/dereplicate operation whose semantics are: given a directory in the hierarchy move the dispersal cut to include it in the replicated portion by doing the appropriate

replicate or dereplicate operations on the intervening directories. Conceptually this can be thought of as traversing the hierarchy and performing the individual replicate or dereplicate operations. Operationally, this function may be quite dangerous, so thought must be given to protecting it suitably.

The other function relates to adding a new host to a configuration. There are a few issues involved with this task. The first is to add it to the configuration database so that it can be identified as running the replication service by other managers. Second, it must be able to get a copy of the replicated portion of the hierarchy from another manager. This is similar to the action required in replicating a directory. In this case one of the Catalog Managers would walk down the root portion of the hierarchy and send copies of each replicated directory to the new host. Since this is presumably done infrequently and at a time before the new host is supporting users, performance and synchronization issues do not seem to be major problems. Finally, the update log file must be reformatted to include the fact that there is a new host in the configuration (i.e., new entries must accommodate the new host in the vector of operation done bits).

A similar inverse set of operations must be done for removing hosts from the hierarchy. The host being removed must be taken out of the configuration database and the log file must be updated to account for the host's no longer participating in the replication service.

10 Input/Output

10.1 Introduction

Devices, such as line-printers, tape-drives, or terminals are integrated into the Cronus system as sub-types of a generalized I/O object CT_IOStream, which supports a generalized set of I/O operations.

We have tried to generalize the input-output operations to make similar operations on different types of objects as similar as possible, so that programs and programmers do not have to be burdened with special-case software which depends on whether the output is a terminal, a printer, a disk file, or the standard input of another program. There are places where these similarities break down, as discussed below. The special-case software is isolated in the PSL so the CRONUS applications programmer will be largely isolated from these details.

10.2 Operations on devices

Devices are objects of type CT_Device, which is a subtype of type CT_IOStream, and implements the standard operations of that type.

Open
Close
(15)
IOLock
Read
Write
IOStreamsOpenBy
OpenStatusOf
CloseProcessOpenIOStreams
CloseAllProcessOpenIOStreams

---

(15) Open and close are used for synchronization. They are also used to trigger those actions that many device managers will wish to perform (e.g., hanging up a modem when the last process closes its output to the terminal, issuing a form-feed when a process opens the lineprinter) when the device gets accessed.

In addition to these operations, device objects also implement a number of special-purpose operations, for example, a tape drive or a disk drive have a Seek operation to allow writing or reading to be done from a particular position in the medium which the device uses. (16) The details of individual device-object operations will be specified as actual devices are added to the CRONUS cluster. (17)

We anticipate a hierarchy of object types, breaking down into finer and finer distinctions. For example, CT_IOStream > CT_Device > CT_printer > CT_lineprinter. Just as there are several kinds of I/O-stream objects, there may be many kinds of lineprinter object, perhaps one for each kind of lineprinter, or there may be page printers and graphics printers.

Device object managers also will commonly refuse a request for "frozen" access. In addition to the exclusivity of access provided by frozen access, one also gains the ability to cancel the writes which have been done to the object. This latter ability cannot be implemented on devices in any meaningful way, so this form of access is not allowed by the device's manager. (18) One may open devices for exclusive access, of course.

---

(16) Other special operations individual device managers are likely to implement are, density and format control for tape and disk drives; many devices may be turned off-line by software; printers will have page-length, page-width, and font controls, and so on.

(17) The description of the special operations on terminal devices is discussed in section 11.

(18) We might at some later date explore making some device managers clever enough to provide their own spooling, in which case one would be able to do frozen writes with the ability to cancel the writes. Such cleverness would likely lead to a number of special-purpose (spooling-oriented) operations, such as "perform output after a specific time", etc. While it might seem that such cleverness is more appropriately placed in a program and not in a device manager, for efficiency reasons one might desire to eliminate the middle-man.

For example, a file to be spooled for printing, the requesting process, and the printer manager may all reside on different machines. There is little point in the data from the file to be passed through the network to the requesting program,

10.3 Implementation overview

For each device object on a host there is a manager for the device. Device managers may manage multiple devices (for example, a host might have only one line-printer manager for all of its lineprinters, or may have a single manager that manages both tape-drives and disk-drives (19)), or a manager may manage a single device. Which of these approaches is taken will depend entirely on the implementation, and is not within the scope of this document. When started, the device manager registers the UIDs for its devices with the operation switch on its host, so that the Cronus IPC mechanism delivers operations on the device object appropriately.

10.3.1 The use of large messages for device I/O

We expect that most I/O devices will be done using a stream interface as supported by Cronus large messages, in order to avoid passing all the I/O messages through the operation switch. This implementation is different from primal files, for example, because of the fundamentally different ways in which we expect the object managers to be implemented. For devices such as line-printers, terminals and tape-drives, it seems realistic to expect that there will be one manager process per physical device. Unlike the primal file system, which is accessed by many processes at one time, an individual device is typically a limited-access entity. Users typically require exclusive access

---

then passed back through the network to the printer manager when the data could go straight from the file to the printer manager in the first place. Thus, a printer-object-manager may implement a "spool for printing" operation which takes the UID of the file to be printed as a parameter. Probably the act of spooling itself should be treated as an object and given it's own UID. Suggested operations on spool-objects: Create (to get a UID for subsequent transactions), Remove (to cancel a spooled action); TimeToBegin (to set the time for the spooled action to take place); as well as the usual printer-oriented operations (header format, font, etc.).

(19) Exotic as this may sound, it is easy to imagine a single manager for DEC-Tape drives and disk drives, for example.

to a device while they are using it. Thus we expect a device manager to be able to maintain a stream connection to everyone who wants to talk to its object. Very few constituent operating systems would permit a process to have so many open network connections supporting the message stream at one time, so we expect I/O from primal files to be datagram-based, rather than connection based. In contrast, I/O from devices may be connection-oriented, bypassing the operation switch for reasons of efficiency.

10.3.2 Reasonable defaults for unspecified options

In order to provide uniformity of access, the device managers assign reasonable defaults for their device-specific parameters (e.g., tape density) if the application program does not issue operations specifically setting them. The goal here is to provide an access mode in which the application program can remain largely unaware of the nature of the object receiving its output or providing its input.

10.3.3 Naming

Devices like any other Cronus objects have names in the globe Cronus symbolic namespace. These names may appear anywhere in the name heirarchy though, as happens on UNIX systems where a similar approach is taken to devices, most devices will probably be gathered together in the directory ":dev". For example, the most popular line-printer may be given the name ":dev:lpt", or devices may be given more descriptive names, like ":dev.fancy_printer_in_graphic_arts_dept", or users may choose to locate the name of a private device in a more convenient directory, like ":usr.melissa.my_printer" (for the printer in Melissa's office) The symbolic catalog name is used only as a convenient means for accessing the device UID and plays no role in the way the Cronus system treats the device. (20)

---

(20) "Attached to" here is taken in a very loose sense. BBN-CLXX has a printer which is physically attached to a BBN-NET TAC port (and which is accessed by a number of hosts), yet it is easy to imagine a device manager for this printer being provided in the

11 User Interface

11.1 Introduction

The Cronus user interface provides uniform, convenient access to the functions and services of the Cronus distributed operating system and the subsystems which run under Cronus. User requests for access to the functions and resources of the system are similar for all DOS components; that is, a request to run a program is the same no matter where the user access point is in the cluster, and no matter where the process that satisfies the request is run.

The user interface includes four major elements by which human users gain access and interact with Cronus to perform tasks.

1. The terminal manager is responsible for the behavior of the terminal or other device by which the user gains access to the system. Cronus supports a number of different terminal managers for users who have a direct connection to the cluster or who access Cronus through the Internet.

2. The session manager controls the user session from login to termination. It operates on the authentication data base (through the Authentication Manager) to verify the user's principal identity, and on the session record data base (through the Session Record Manager) to record information about the session. It also creates parallel execution threads and allocates portions of the terminal, under user control, to each thread.

3. The command language interpreter (CLI) receives requests from the user to create processes and execute programs to perform the tasks.

4. The user programs or applications that actually perform the tasks run in program carriers (see Section 5). The terminal manager, session manager, and the CLI cooperate in creating these program carriers, loading them, passing

---

Cronus cluster.

parameters to them, and directing the input and output to the places that the user has requested.

The design of the Cronus user interface has been influenced by the following considerations:

- The user interface should deal effectively with the distributed character of the operating system.
- Variations in cluster configurations and in user requirements will likely lead to a number of different user interfaces, and these interfaces will evolve. Therefore, the current implementation should focus on the underlying structural concepts needed to support a variety of presentation methods.
- The utility of Cronus depends on widespread accessibility. Therefore, the initial implementation should support commonly available terminals instead of more powerful devices which are now just becoming available.
- The user interface should support system reliability and error recovery from malfunctions during a user session.

The consequences of these observations for the design of the user interface in a distributed system are explored in the next section. The terminal manager, session manager, command language interpreter, and the pattern of the cooperation among them and their use of other system objects are discussed in the following sections.

11.2 User Interface Design for a Distributed System

The Cronus user interface is a generalization and extension of user interfaces provided by other computer systems. Since Cronus is a distributed operating system that integrates a collection of otherwise independent computer systems, the implementation of a function may be dispersed across the cluster. The Cronus user interface is independent of the user interfaces for the COSs.

The following are some of the design objectives for the user interface that have been influenced by the distributed nature of Cronus.

1. Command invocation and program control should be uniform across the cluster.

2. Multiple parallel activities should be supported directly by the user interface.

3. The user should be able to start and control distributed activities.

4. System operation should be independent of the location of the terminal manager, session manager, CLI, and user processes.

5. The user interface should support detection and recovery from malfunctions affecting only parts of a user's session.

6. The user should be able to issue commands directly to the COS.

First and foremost, Cronus itself provides for the uniform invocation of any command. The command interpreter finds the command in the Cronus symbolic catalog and creates a program carrier for it. Because the symbolic name space is host independent, commands can be organized in any manner convenient to the user, for example, all the programs used to carrry out a particular task can be cataloged in a private directory, even if some of them can only be executed on specific host types. The host is normally selected by examining the type of the executable file for the command

A Cronus cluster may have more than one host of a particular type, and different copies of reliable files are stored on different hosts. The interface allows (but does not require) the user to communicate an intention to use a specific instance of any replicated resource.

A single user session may contain a number of independent tasks executing in parallel on different hosts. In such a session, the user can exploit the true parallelism which separate processing elements provides and reduce the effects of communications delays by selecting the host on which a task executes. Cronus provides device-independent mechanisms that support the use of a single terminal for controlling parallel activities. The effectiveness of a particular terminal for this purpose is, of course, dependent on the capabilities of that device.

A programmer can develop multi-part applications in which the individual parts (program carriers) can execute on different hosts. To the end user, the distribution of components can remain largely invisible, since the programmer and Cronus can take care of the details of the distribution. In particular, a task may consist of a multi-host pipeline of processes, in which a process running on one host can pass its output directly to the input to a process running on another host.

The Cronus architecture provides several kinds of access point. Although the user interface has comparable components for each of these access points, the location and mode of interconnection among the components will differ. The decomposition of function in the user interface permits flexible distribution of these components.

On the other hand, the distribution of the components increases the cost of synchronization and probability that a single host failure will affect the user session. To reduce synchronization traffic, Cronus does not maintain a centralized record of all elements in a user session. Rather, this data is distributed among the managers responsible for the individual parts. This makes the interface somewhat tolerant of failures and provides a basis for the design of a reliable user session

The user interface facilitates direct access to COS functions through a user Telnet function, which can access the COS command interpreter for the hosts of the cluster. Telnet is treated as a parallel activity with other user activities; that is, it is a separate thread in the user session.

11.3 Overview of a User Session

A session begins when a user activates a terminal that is connected to Cronus and proceeds with a system login. The session normally ends when the user logs out. During the session, the user interacts with the system to run programs which interrogate and manipulate Cronus resources and to perform such job specific functions as word processing or data base inquiry.

Users gain access to Cronus in one of following ways:

1. Terminal access controllers (TACs). A Cronus TAC is a terminal multiplexer connected directly to the local area network. Cronus TACs are implemented in dedicated GCEs.

2. The Internet. The Cronus local network is connected to the Internet by means of an Internet gateway. Users outside the cluster may access Cronus through the standard terminal handling protocol (Telnet) which operates upon a lower level, reliable transport protocol (TCP).

3. Mainframe hosts. Cronus mainframe computers can have terminal ports that enable access to Cronus.

4. Dedicated workstation computers. A workstation is a computer that is, at any given time, dedicated to a single user. Workstation hosts have sufficient processing and storage resources to support non-trivial application programs, such as editors and compilers, and to operate autonomously for long periods of time(21).

The user interface has four principal modules: a terminal manager, a session manager, the session record manager, and the command language interpreter

When the user activates a terminal, the terminal manager connects the user to a session manager. There is a session manager for each active user. It has a limited set of commands for initiating and manipulating sessions and session data (see

(21). The Primal system will not support workstations.

Cronus User's Manual session(1)). The login command, which initiates a new session, performs two basic functions. First, it identifies the user, establishes the access rights for the session, and gets the user data needed for session initialization. Second, it creates a session and records it in a session record. A complete description of the session is distributed among a number of system components, but the session record object records the existence of the session and certain other key items (see Cronus User's Manual session(3)).

After the session manager has identified the user, it starts the initial subsystem specified in the user's principal object (see Cronus User's Manual principal(3), principal(4)). This can be either a general purpose command interpreter or a special purpose application. The principal object may also request that the initial subsystem be run on a specific host.

The session manager maintains session data as part of its temporary state, that is, this information does not survive if the session manager crashes. The session record manager, on the other hand, maintains the basic information needed for session recovery in non-volatile storage.

The initial subsystem runs in the first processing thread in the session. The user may create more threads, each of which consists of a varying number of program carrier processes organized into a hierarchy rooted at the program carrier created by the session agent. This program carrier is called the head process of the thread.

Often the head process is a command language interpreter (CLI). This is a program that interacts with the user to receive commands, which it performs by creating and controlling processes. In the following discussion, we assume that the head process of the current thread is the Cronus standard command language interpreter, which is called cli (see Cronus User's Manual cli(1)).

The head process can execute a command that terminates the thread. The session agent may also force the termination of a thread. The logout command terminates a user session. At the end of the session, the session record object is removed, and the terminal is free to support a new session.

Instead of executing logout, the user may detach from the session and re-attach to it later. Processes in a detached session are no longer controlled by the session manager and from the terminal. These processes will continue execution until they require terminal input or output, at which point they will block, and wait until they are re-attached. When the user re-attaches to this session, the new session manager and terminal takes over as the source of control and terminal input/output. The session manager command resume causes the processes to continue. This procedure is also used in recovering a session which has been detached by a host crash.

The user interface assigns the responsibilities for user session activities as follows.

- The terminal manager encapsulates the physical terminal device. It handles the terminal device, directs the keyboard input to the active process, receives the output from one or more active processes, and manages the display (for video display units)

- The session manager initiates user authentication, creates a thread, starts the inital subsystem, creates and manages additional threads, attaches and detaches sessions, and assigns terminals to processes.

- The command language interpreter reads and parses command line input, starts and controls processes that run the commands, and controls assignment of standard input and output.

- The session record manager creates and maintains records for active and detached user sessions.

In addition, other components of Cronus support the user session, of particular importance are the program carrier manager, the catalog manager, and the authentication manager.

11.4 Terminal Manager

The terminal manager is the process which is closest to the user. It provides the Cronus interface to the physical device, through cooperation with the COS of the host to which the terminal is connected. The terminal manager supports three broad classes of device.

- o hardcopy terminals that are strictly line-at-time devices capable of producing upper and lower case alphanumeric characters and the standard ASCII control character set;
- o ASCII video terminals (often called CRT terminals or video display units) that support cursor addressing on a display screen that is large enough to support, for example, a full-screen editor, and
- o advanced terminals (often called bit-mapped terminals) that contain a processing element and enough memory to support multiple display areas and graphical output.

The primary focus of the primal system is on the ASCII video terminal because there are many of them available today. Cronus supports the sharing of a single, physical terminal device among the parallel activities in a session. This terminal multiplexing will be most effective when an advanced terminal is available, but will be possible in a limited fashion with the other terminal types.

The terminal manager encapsulates the physical terminal, the corresponding Cronus object is of type CT_Physical_Terminal (see Cronus User's Manual phys_term(3)), which has a number of subtypes corresponding to the different kinds of terminals. One or more objects (called Cronus terminals or simply terminals in the discussion below) of type CT_Terminal is associated with each physical terminal. This provides a mechanism for multiplexing or sharing the physical terminal among a number of Cronus terminals. The Cronus terminal is the input/output device for a process. Since terminals are Cronus objects, they have all of the usual properties of objects, including host-independent access. In addition to the generic operations defined on CT_Object (see Cronus User's Manual object(3)), the following operations are defined on objects of type CT_Terminal.

Open
Close
Read
Write
Activate
Deactivate

Programs may treat a Cronus object of type CT_Terminal like an ordinary terminal, since it has a keyboard and a screen, although either or both of these may be inactive at any time. Each thread in a user session, and the session manager itself, has its own object of type CT_Terminal, which will simply be called the terminal in the discussion that follows. Within a thread, processes coordinate their access to the terminal through the terminal manager.

If the physical terminal supports independent display areas (windows), the session agent maintains a window for status displays. The rest of the physical display contains one or more regions, each of which is used for the output of a single terminal. The physical keyboard can be associated with only one of the terminals at any time; the thread that owns this terminal is the current interactive activity in the session. The keyboard can be transferred to the session manager's terminal by a control character sequence (see Cronus User's Manual terminal(1)). Once the session manager is in control, the user can execute commands to create new terminals, remove old terminals, and change the current interactive terminal (see Cronus User's Manual session(1)).

Output to any of the regions currently displayed is immediately visible. Input is directed only to the current thread. Normally all input characters go to a single process. However, when one process creates another process, it may request certain (control) characters to be intercepted and sent to it; the interrupt facility discussed in Section 11.8 is implemented using this facility.

Processes invoke Read and Write operations on the terminal to get input from the keyboard and write to the display. These use large messages of indefinite length to provide a stream between the terminal manager and the process. A process will have two messages associated with the keyboard; it sends read requests on one of them, and receives the input on the other one.

As keybord input is collected, it is used to fulfill any outstanding read operation. Since the terminal is shared among the processes of the thread, characters are sent only in response to a read request. If there is no outstanding request, the terminal buffers characters until it exhausts the space allocated for them.

When control of the keyboard is transferred from one process to another, the old process stops issuing read requests. If the new process needs keyboard input, it establishes the two messages used for the stream and begins issuing read requests of its own. The PSL routines for reading and writing take care of the details of establishing the messages, so ordinary applications need not be concerned with them. The Write streams are not controlled; simultaneous output from two processes in a thread may become interleaved unless they are coordinated by the application program logic.

Each terminal has mode settings which control its behavior. These are discussed in detail in the Cronus User's Manual page terminal(1). Among the most important are the following.

1. Read activation termination character set. An input character from this set terminates the current read request. The terminal manager stops sending characters after it transmits the ones it has, including the termination character, until it receives another request.

2. Echo control. Input echoing at the terminal manager may be either on or off. If it is on, it may be performed immediately or deferred until the characters are used to satisfy a read request.

3. Buffering and local editing. Terminal input may be buffered until an activation request termination character is typed. If the input is buffered, local editing functions are also available. If the input is unbuffered, it is sent as soon as it is accepted when a read request is active, the application process then assumes the responsibility for editing functions.

4. Interrupt character handling. The user may set certain characters as interrupt characters, see the discussion in Section 11.8.

11.5 Session Manager

The session manager creates and removes user session records, controls the allocation of the physical terminal display, and creates and controls threads.

During a simple session, in which a user executes a series of commands sequentially, the session agent is largely invisible to the user. The user may, however, wish to initiate and control parallel activities. Each collection of parallel activities is a thread. Session threads are objects of type CT_Thread. At any time during the session, the user can instruct the session agent to create additional threads which operate in parallel with other existing threads(22). A thread can be used to support parallel processing or to maintain the state of some activity while the user shifts attention to another activity.

The first process created in a thread is called the head process, and is usually a command language interpreter. The default head process is an instance of the principal's initial subsystem, but the user may select any program in the Cronus symbolic namespace.

A new thread is created whenever a Telnet connection is opened, with the Telnet process at its head. The connection may be to any Internet host, either within or outside the cluster. For the forseeable future, Telnet paths to cluster hosts will be needed to support activities not yet incorporated into Cronus, such as maintenance of the COS.

The following commands are supported directly by the session manager (see Cronus User's Manual session(1)):

- Start a new session (login)
- Terminate a session (logout)
- Attach session agent to an existing session (attach)
- Detach session agent from an existing session (detach)
- Initiate a parallel activity (create_thread)
- Terminate a thread (killthread)
- Create a Cronus terminal (make_terminal)

---

(22) There is user-settable control key that activates the session manager so the user may invoke session manager commands.

- Remove a Cronus terminal (remove_terminal)
- Map thread to region (map_thread)
- Display threads (showthreads)
- Activate named thread (thread)
- Telnet to host (telnet)

11.6 Session Record Manager

The session record manager maintains the centrally accessible, non-volatile record of active Cronus sessions in objects of type CT_Session_Record (see Cronus User's Manual sess_rec(3)). A session record object contains the following data.

- Session UID
- Creating principal
- Time of Creation (for identification purposes)
- Lists of thread UIDs
- ACL
- Session agent ProcessUID

A session record is created at the beginning of each Cronus session. During the session's lifetime, data is added and removed by the session agent. The session record is used in recovery after a host or system crash.

The session record can be accessed by other programs to report about an individual session or all current sessions. In addition to the generic operations, the following operations are defined on objects of type CT_Session_Record.

- Read_Public
- Read_Private
- Write_Session_Record
- Lookup_Principal

11.7 Command Language Interpreter

A user request usually consists of a command name plus one or more parameters or arguments. There are three basic kinds of arguments for cli: names of objects from the Cronus catalog; control parameters, called switches; and application-specific parameters. Switches may be associated with either the command as a whole, modifying its behavior, or with one or more of the object names that appear on the command line.

If one thinks of the command as a series of words typed on a line, the command name is the first word. The command name specifies the action to be performed; the actual name is often a simple English word suggesting that action, for example, print. Cli interprets the command name as an entry in the Cronus symbolic catalog. It expects the command name to be the symbolic name of an object of type CT_Executable_File. Either a complete or partial pathname may be entered on the command line. A designated set of Cronus directories (called the search path) are used to resolve partial pathnames; the first match encountered causes the search to stop.

There is a small set of commands built into cli. These are used to control the command interpreter's environment (such as the current working directory) and the execution sequence of commands. Executable objects may be either process images or files containing commands. The built-in commands that control execution sequence are most often used in command files.

The executable object may be augmented by a syntax definition, so the command interpreter can know the number and type of the arguments, default and legal values for the switches, and help information for the command. Users may associate private syntax definitions with public commands. Commands which have syntax definitions, either private or public, are called defined commands.

Command arguments are passed as part of the process descriptor (see Section 5 and Cronus User's Manual process(4)) of the new program carrier process. When the command syntax definition is available, cli performs type and range checking on parameters, and conversion from alphanumeric to internal representations for certain of types, including Cronus object name and integer. Both forms are passed to the application

process, since the character string form is of use in some cases, for example in generating error messages.

The syntax definition facility is particularly valuable in a distributed environment for the following reasons:

- The cost of remote command invocation is generally higher than it is in monoprocessor cases. Parameter error checking reduces the frequency of execution failures caused by usage errors.

- If the command interpreter knows the names of some of the objects that the command is operating on, it may be able to use object location as one criterion in its selection of a site for command execution.

Many command arguments are cataloged objects. Cronus supports a working directory list, which is an ordered collection of directories that are used in relative pathname searches for named objects. The user may change this list at any time. The cli also supports partial name recognition. The user may press a key to get a list of all matches for the partial name, using both the working directory list and the standard wild-card facilities (see Cronus User's Manual sym_name(4)) of the Cronus catalog, from which the actual name may be chosen. There is also a deferred recognition key which allows the user to ask for the matching to be done, but not reported interactively.

The help key can be used to display help information which is found in the syntax description of a defined command.

The command interpreter allows a user to provide a host designator when specifying an object name, including the name of the command itself. For example,

```
edit textfile@CVAX
```

would invoke the editor on the copy of textfile stored on the Cronus VAX.

```
copy file1 file2@GCE3
```

would make a copy of file1 under the name file2 and store the new

file on host GCE3, and

```
Radar@CLXX other_parameters
```

would select host CLXX to run the subsystem Radar.

Objects of various types may be cataloged in the Cronus symbolic name hierarchy without restriction. Often, a user will wish to select objects of a specific type, so a standard switch is defined for type designation. As an example, a user would type

```
dir_display file_name.*/type=reliable_file
```

to display the names of those objects in the current working directory list that match the wildcard pattern file_name.* and are of type CT_Reliable_File.

Cli performs two kinds of initialization. First, internal variables are set from a profile data file, which consists of lists of (name, value) pairs. This file can be maintained using edit key value (see Cronus User's Manual editkey(1)). Second, cli executes a profile command file.

After cli has collected and parsed a command, it creates a program carrier object, loads it with the executable image and starts it. Normally the process uses the same terminal as the command interpreter does. Therefore, cli releases control of the terminal to the user process, and waits for it to terminate before collecting another command.

Cli uses the program carrier process support for input and output redirection (see Section 5 and Cronus User's Manual prog_carr(3)). The redirection is indicated by the punctuation character >, thus the command

```
dir_display file_name.* >newfile.lst
```

would place the result of the catalog lookup of file_name.* in the file newfile.lst. When cli redirects output into a file whose name did not previously appear in the Cronus catalog, it creates a new primal file. The user may use the standard switch (/type) to designate another type, for example,

```
dir_display file_name.* >newfile.lst/type=reliable_file
```

will create a reliable file to receive the output.

The user can specify that two or more commands should be executed simultaneously and linked together linearly, in such a way that the output of the each command becomes the input to the next one. We refer to the collection as a pipeline. Since the components of a pipeline may be on different hosts, the user can dynamically construct multi-machine distributed commands.

11.8 User Processes

In most cases, actual work of an application is carried out by a user process that is created in response to a command issued to cli. These user processes are program carriers, and make use of all of the properties of those objects. Objects of type CT_Program_Carrier have been discussed in Section 5.5. Application programs typically make extensive use of the PSL. In this section, we discuss interrupts and user error reporting, both of which are supported by the PSL.

Sometimes a process needs to be terminated by an interrupt or signal. Cronus supports two forms of interrupt, a hardkill, which terminates the process immediately without giving it the opportunity for application-specific termination processing, and a softkill that gives the application process the opportunity to terminate cleanly. In the event that programs do not respond to softkill requests, hardkill can be imposed. Interrupts are usually invoked by typing a control sequence during a user session, but they are also generated by a command (see Cronus User's Manual signal(1)).

Programs may choose to receive softkill signals, and use them for application-specific purposes unrelated to process termination. Cli will always receive the hardkill signal and remove the application process.

Interrupts invoke the Stop operation on program carrier objects. The exact implementation on a particular host depends on the facilities of the COS that are available to the program carrier manager.

The processes created by cli form a hierarchy of program carrier objects, which may be decomposed into sub-hierarchies of the thread object. Any subtree of the thread hierarchy is called a process group. An entire thread is the largest process group. Process groups are managed by the program carrier manager in the current implementation. Operations on process groups support convenient control and cleanup of process subtrees.

Methods for reporting errors in Cronus are designed to support a variety of program structures and execution environments. There are two basic program structures:

> Asychronous processes, often called manager processes because object managers are of this class; these processes receive messages from a number of sources and may not wait if they issue requests to other managers to satisfy incoming requests. Error handling in manager processes is discussed in Section 4.6.
>
> Synchronous processes, which process data that arrives in a more or less predictable fashion, often from a terminal or a file. When these processes send messages, they usually wait for a reply.

We have identified the following execution evironments:

> Independent processes are asynchronous processes, particularly object mangers that are daemon processes started by the Monitoring and System or by another daemon process.
>
> Interactive processes may be either synchronous or asychronous. In this environment, a human user carries on a conversation with the process. Examples of processes in interactive environments include the traditional applications of distributed systems, multi-host database systems, office automation, and program development systems.
>
> Pipelined processes consist of two or more programs which might normally be run in an interactive environment that are connected in such a way that the upstream process writes its output on the input of the downstream process. A pipeline can span host boundaries

Background processes are generally interactive programs which are set into execution in such a way that the data which normally comes from the user is found somewhere else (usually in a file).

In the interactive case, where the error is reported directly to the user, we have a situation that is similar to the one in an ordinary, centralized operating system. It can be seen that error handling is similar in pipeline and background cases.

A program in an interactive environment will also report certain errors to the Monitoring and Control System (MCS). These include errors caused by system resource limitations and some kinds of access control violations.

Independent processes, including Cronus managers, report errors to the client which issued the original request, and may also send a message to the MCS. In addition, Cronus managers keep statistics on the kinds of errors which have been detected, and report them to the MCS periodically.

The responsibility to terminate or continue processing belongs with the application or manager, so PSL routines never take pre-emptive action, and never terminate the process. The PSL routine cannot understand the situation well enough to exit properly, since the routine may be executed within an atomic transaction, or within a composite action which has other work-in-progress entries (see Section 4.6). Instead, it sets parameters describing the condition in an error block (see Cronus User's Manual error(4)), and the application error handler fields the error and processes it.

The standard error list may be found in the general Introduction to the Cronus User's Manual. Each PSL routine page in Section 2 of the Cronus User's Manual lists the errors which may occur during the execution of the the function. In most cases, an interactive process would perform any necessary cleanup, and then use the standard error reporting routines (see Cronus User's Manual error(2)).

Whenever an error is detected in processing a request from a client process, the error condition is reported through the reply message. The error procedure uses the standard message

structure, and certain assigned keys. When it is necessary to report an error to the MCS, the process uses a standard routine to generate the message to the MCS (see Cronus User's Manual error(2)).

12 Monitoring and Control

12.1 System Capabilities

The monitoring and control system (MCS) for Cronus includes monitoring and control of hosts and of the Cronus functions on these hosts, of the network substrate, and of gateways. 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. Where practical, it also monitors and controls Constituent Operation System (COS) functions from the same station, but such functions are limited by our desire to modify the COSs as little as possible. The discussion in this section includes elements of the Reliable System as well as of the Primal System. These additions are included to assure that the Primal System design does not interfere with future extensions.

Cronus is restarted from the Monitoring and Control System. For some hosts, the MCS will invoke functions already on the hosts, in other cases (for example, GCEs which have no disks), the MCS will download the host to start Cronus.

Network monitoring and control of a local area cable-based network such as the Ethernet is relatively simple. It includes a detection and reporting of changes in host availability, monitoring and controlling traffic levels on the cable. Cable utilization and the traffic level of each host is measured. Priority or allowable traffic density may be set for each host. Transmissions from a host may be stopped altogether.

12.2 System Model for Monitoring and Control

Cronus consists of a set of services(23) and low-level system support entities, including the Cronus IPC mechanism. The

(23) A Cronus service is a process which performs Cronus operations in response to requests from other Cronus processes. All object managers, for instance, are services.

MCS is a set of processes on a Cronus host, its functions can be executed from anywhere in the cluster.

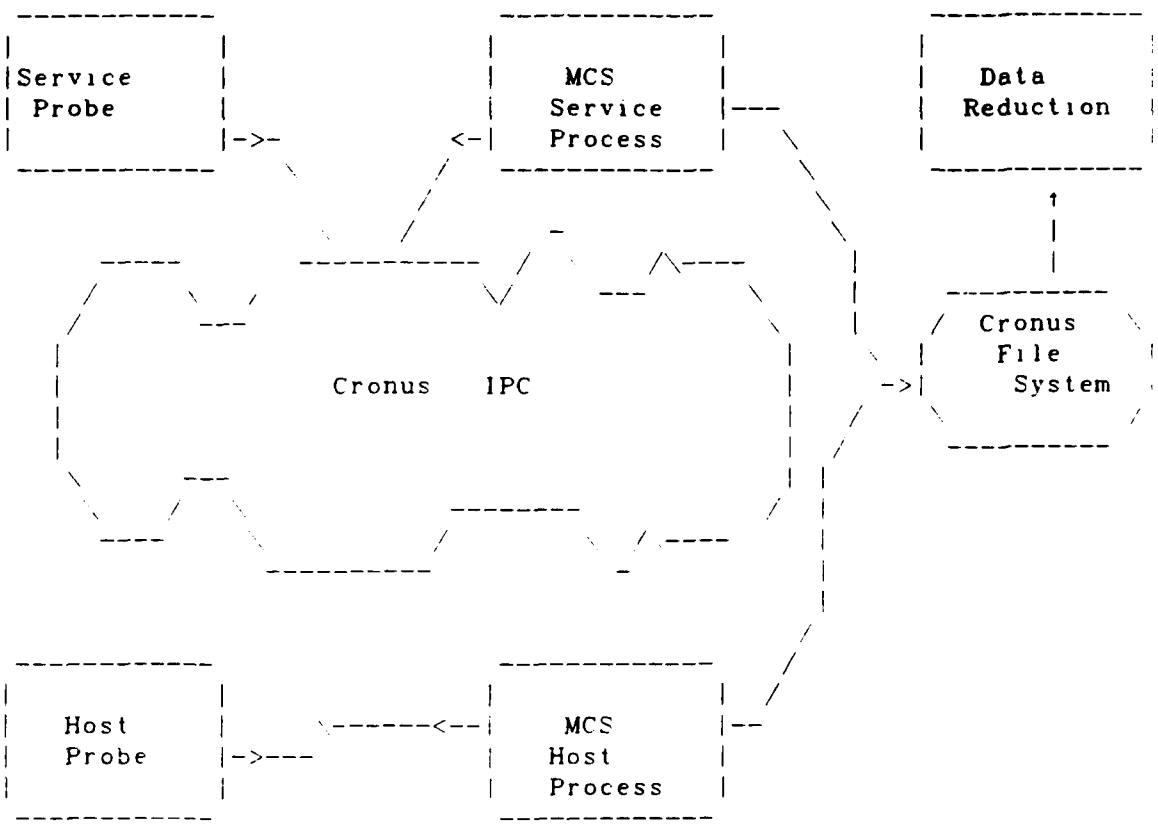


Figure 12.1 Structure of the MCS

The MCS monitors both the support layer and the services. The set of services is extensible, and the MCS is designed to accommodate new services.

The MCS is based on a functional decomposition rather than on a site-based decomposition of the system. For example, one service monitor monitors all file system managers while another monitors authentication managers. The MCS will be aware of distinctions between sites and to distinguish them in its reports.

12.3 Structure of the MCS

The MCS runs as one or more Cronus processes. The MCS station is not bound to any particular site, although certain information gathering functions are most conveniently performed at one location. It uses the Cronus file system, in which it will store data, and the Cronus IPC facility. The MCS will be divided into two parts. The first part is the interactive section, which does on-line data collection, display, and control of Cronus. It obtains status information from host and service probes, and incorporates it into its own data base. The second part performs data reduction and generates reports.

The interactive section of the MCS consists of a very low-level module and a higher level module (see figure 1). The majority of the MCS resides in the high-level module, a Cronus service which communicates with its probes through the Cronus interprocess communication facility. The low-level module uses only the lowest level of network protocol (User Datagram Protocol). This primitive lower level can be relied upon when little of Cronus is functioning. This portion will be implemented first. It provide the functions required to bootstrap Cronus, to examine and alter memory on Cronus hosts, and to do simple monitoring of the Cronus network.

There are two types of reports to the MCS: polled messages and traps. Polled messages are reports in response to a request from the MCS. Traps are reports from probes which are unsolicited. They normally represent unexpected or unusual events.

The MCS uses polled messages as the primary data gathering technique. The polling request provides a mechanism which will quickly recognize when a host or service disappears.

Traps are employed for reports about specific events, which may require real-time response, or which are unanticipated. For instance, the crash of a service would be reported as a trap, so that service restoration or reconfiguration could be instituted immediately. A host coming up would similarly be reported by a trap message, because of the timeliness of the information and because a new host on the network might not get any unsolicited

polls(24).

The MCS contains a trap logging service. Trap reports are generated by both host and service probes. Trap messages include a service type and priority in their header, so that display routines can easily determine which traps require immediate display in a high-priority window, and so that the operator can easily select all traps in a priority range from a given service class (e.g. file system). The trap logger could be extended to permit automatic operations in response to traps, so that a "service crashed" trap report could be used to force a restart of the service from the MCS.

The display processes normally directs critical reports to the system operator, with each process controlling one or more text streams. A text stream may be directed to a display terminal window, a hardcopy output device, a file, or several different places. The operator terminal should support a multi-window display, which will enable the operator to monitor a variety of aspects of system operation simultaneously, with one window usually reserved for critical reports. Other windows will be created to present data as requested. For instance, an operator might choose a process in one window which presents the general status for all hosts in the network, and another window to present the load status for a particular host of interest.

When the sophisticated window package is not available, a simpler interface would enable the operator to monitor one window at a time, the difference would be invisible to the MCS since each window would look to it like an independent display.

The data reduction facilities of Cronus can reside wherever convenient, and will be regarded as background tasks. The integrity of the system does not depend on their availability, but their reports should prove useful to the tuning and management of the network.

---

(24) Polling for hosts which are known to Cronus but currently down would continue at a low rate, however, so that a lost trap for such a host coming up would not be fatal.

The data reduction section will take advantage of the fact that the files generated by the interactive section are available globally as part of the Cronus file system.

12.4 Host Probes, Service Probes, and Network Monitoring

A host probe is a primitive entity which every Cronus host must provide to report status to the MCS. A host probe must at least report the presence of the host and its internet address at the time the host operationally enters Cronus, and must respond to AREYOUTHERE messages broadcast from the MCS. The host probe is the distributed part of the low-level section of the monitoring and control system. A host probe will often offer further information in its report. host type, probe reports available, current MCS reports, Cronus services, level of integration, etc.

Service probes are monitoring entities in all Cronus services. Services to be monitored will include object managers, terminal concentrators, and user authenticators. Service probes reflect a functional rather than site-based decomposition of Cronus. Data from related service probes on different hosts are combined in the MCS, in order to present a more understandable picture of the service. The MCS specifies what types of data should be collected and reported through poll responses and through traps.

A service probe is located within the service. Unlike host probes, they may require a certain level of Cronus functions, since the loss of service monitoring and control does not compromise our ability to restart the system. Service probes use the full range of Cronus services, especially the Cronus IPC facility.

Some messages, including control messages and high-priority monitoring, will run with a priority above that of the service. Most monitoring, however, will run with a priority below that of the service itself.

The service probes for the Cronus file system reports the loading on the local portion of the file system, the number of requests for various classes of services, etc. It may also

include the ability to trace all activities on particular files (using traps) as a debugging aid.

The process manager probe reports machine process loading, both for Cronus and non-Cronus processes, and optionally supports tracing services for activities on Cronus processes. The probe will report certain classes of exceptional events on processes, and will provide services, invokable from the MCS, for invoking and killing processes, and for tracing process activity on a per-process basis.

Gateway monitoring would normally fall into the category of service monitoring; however, the gateway already reports status in response to polling by a host. We will use this capability to obtain gateway and internet status. Since we do not wish to do development in this area, we will to restrict ourselves to the available capabilities.

The MCS will not monitor the cable network traffic directly. Rather, it will gather reports from hosts on the traffic sent, traffic received, and the collision rate at each node.

12.5 Loading and Debugging Support

The control function has the capability for restarting Cronus on the hosts of the network. It may do this in one of two ways. In some cases (e.g. GCE), this includes transmitting the code directly to the host to be loaded. In other cases, the computer s own loading sequence is invoked, using its private secondary storage. In no event should the downloading procedure require the assistance of a third machine. Some machines may detect some of their own failures and restart themselves

A distribu ed, heterogeneous system such as Cronus poses special problems for debugging tools. The goal is to have a sophisticated debugger which runs on one host and debugs on another. We would like to have a single debugging system be capable of debugging computers of differing architectures. Moreover, we would like the debugger to be able to debug at source language level to provide for efficient development. Currently, the leading candidate for developing such a tool is

XMD, which is adapted from the multi-window editor PEN. XMD does not currently debug code in high-level languages, but can be extended in this direction, since it does not depend on the structure of the debugged code, relying instead on symbol table entries to provide it with information about the target code. XMD may soon be extended to debug C source code as part of the effort of another project at BBN.

12.6 Cronus Initialization

The initialization of Cronus is performed from the Monitoring and Control Station. In initializing the system, the MCS will have no certain knowledge of what hosts are available. The first step is to poll for the available hosts, and then to initialize each host which responds.

The initialization of Cronus proceeds as follows(25). (See the scenario in Section 13.)

1. The MCS broadcasts AREYOUTHERE onto the network.

2. Each host has a routine in its COS that listens for AREYOUTHERE and responds with HEREIAM and the parameters (a) name, (b) internet address, (c) boot class, (d) boot file name, and any other required information. The name is printable. The boot class indicates the method used to initialize the host. Class 1 hosts accept a BOOTYOURSELF command and initialize local Cronus software upon its receipt. Class 2 hosts require a BOOTLOAD command, which is followed by a boot file (item d) which passed to the to the host with the code to load. Class 3 hosts require a host-specific loading protocol, which is executed on the MCS from the boot file. (There are no plans to

(25). These messages do not use the full Cronus IPC mechanism in the first four steps of the procedure, since the operation switch and primal process manager are not in place on the host being initialized. Instead, they will be implemented as VLN messages.

implement Class 3 hosts in the ADM.)

3. When the MCS receives a HEREIAM message, it enters the addresses of the host in a host monitor table, with a notation that the host is not up. It then sends a BOOTYOURSELF message if it is a class 1 host, or a BOOTLOAD followed by the required file if it is a Class 2 host.

4. When a host has completed Cronus initialization, it sends a message BOOTDONE to the MCS. Alternatively, it may send the message BOOTFAIL, possible with parameters indicating reason (e.g. "missing file block 5"). The MCS may then retry the boot, if appropriate.

5. After the host is initialized, the MCS will communicate with it using the Cronus IPC mechanism. It will normally obtain a list of available services and will then ask it to start up the services it supports.

The initialization procedure requires a small amount of code resident in each processor in order to respond to the MCS messages. This code will fit in ROM on machines which do not have secondary storage.

12.7 Siting the Monitoring and Control System

Should the MCS be located on the GCE or on an application host? Using a GCE is desirable because it can be specially configured to support the MCS; it is intended to be the dedicated processor, it provides controlled, predictable performance with dedicated, low cost hardware, and it is expected to be redundantly available. Since UNIX hosts may not be available redundantly, we would less often have back-up service if use it on a UNIX application host for the MCS. On the other hand, building the MCS on an application host has several advantages. the UNIX host provides a much richer development environment, have already been written for UNIX, so that less program development would be necessary, we can take advantage of the set of available UNIX utilities.

For the near term, we will build the tools on UNIX. We will be careful to code the routines in a portable manner, so we can easily move them to a GCE environment. This provides us with the benefit of using UNIX in the short term, while keeping the eventual goal of relying on redundant GCE's for Cronus services.

12.8 Phased Implementation

Implementation of the monitoring and control station will occur in phases, both in terms of functionality, and in terms of reliability and performance. The functionality will be increased both as the reporting capabilities of the probes increases, and as the need for data analysis grows.

Initially, the MCS will exist on a single host, without strong reliability or performance goals. We will first build the host monitoring section of the MCS, and simple host probes in order to be able to start and restart Cronus hosts and services, and to record the status (up/down) of hosts. As services are written, we will add service probes, and extend the MCS to monitor them. This initial system will utilize the UNIX file system until the Cronus primal file system is available, and will then convert to the use of Cronus files. Later the MCS will reside on a GCE and will use standard Cronus files.

```
* Revision 1.1  83/06/06  10:39:29  bjw
* Initial revision
*
```

13 Scenarios of Operation

13.1 Basic User Commands and Functions

This section presents examples of the use of Cronus functions and of the integration of structural units. Scenarios are presented for typical system and application tasks. The intent is to suggest the interactions through the flow of control and shared data. The scenarios also suggest how the primitive functions might be combined to support operations required of modern operating systems. The first few sections are narrative, and the later ones provide pseudo-code examples. Details of syntax and calling sequences in these examples are not those of the actual implementation.

Many of the user commands and functions of Cronus fall into the following categories:

- Session initiation and termination. Login, Logout, Attach, etc.
- User and system data base status and maintenance. Display and edit user records, access control lists, show logged on users, etc.
- File manipulation and file/directory maintenance. name lookup, read, write, directory listing, etc.
- Program invocation and control. create process, terminate process, etc.
- Input/Output: List file etc.
- System Operation. Starting the system, monitoring its components, etc.

Each of the following sections presents a scenario from one of these categories.

13.2 Registering a New User

New users may be added to the system only by members of the administrative group. The command to create a principal entry issues an Invoke operation specifying the logical name for the principal data base manager (CL_Principal) as the target process, and including the Create_Principal operation and its parameters in the message text. The Invoke uses the Locate(CL_Principal) operation, to find an available principal data base manager, then sends the message text to one of the sites that responds using SendToHost. The site identifier may be cached to simplify subsequent requests. The principal data base manager creates a user entry and returns the unique identifier for the new object. This UID is the Cronus internal name of the principal, and will appear in Access Group Sets and Group specifications. It may also be used to identify the user record whenever that record needs to be accessed.

When a principal is added, a number of user data base entries are initialized. One of those is the priority range authorized for the user. A private directory is created, and the principal is given all rights to it. The pathname for this directory is entered as the default home directory for the principal. The home directory serves as the repository for command interpreter profile data that specifies user-customizable system features.

13.3 Login

A user may connect to Cronus either through Telnet and a standard session agent running on a shared Cronus host, or through a Cronus Terminal Access Computer (TAC). Telnet supports access from outside the cluster through gateways, and from other devices obeying the protocol.

Access through a Cronus terminal device process is available only from a host that supports Cronus interprocess communication protocols and will probably be supported only on workstations or Cronus TACs. It is more powerful, because the access point software is fully integrated with Cronus.

To initiate a session, a user must have a terminal device process to manage his terminal communication, and a session controller process to manage interactions with the system. Telnet access requires both processes to execute on a shared host of the system. A workstation access path can support both processes; a Cronus TAC access path places the terminal device process in the TAC and the session controller process on a shared host.

Login is handled by the Cronus session controller process. The user is prompted for a login name and password, which are used by the session controller process to build a request to the Authentication Manager by invoking the operation.

```
Authenticate_As(name,encrypted_password)
```

On receiving this message, the Authentication Manager retrieves the associated principal data base entry, verifies the password, and creates the Access Group Set for the process.

The Authentication Manager interacts with the Cronus Session Manager to record the session. The Session Manager assigns a session identifier and adds it to the table of active sessions. A session record contains are the UIDs of the session principal, controller process, and terminal device process. This table is used to satisfy status requests about the cluster and active users. Some emergency procedures, (for example, destroying all processes associated with a session), may also rely upon this table.

The session identifier, the AGS, and other user data base entries are placed in the process environment through an interaction with the process manager for the authenticating process.

After modifying the process environment to indicate successful authentication, Authenticate_As returns the principal UID to the authenticated process. This identifier is used to interrogate the user data base for other information needed to complete the login sequence. One such item is the default home directory, the symbolic name of the initial Cronus directory which is used for unrooted catalog lookup operations, including the search for additional user initialization data. The directory name is converted to a catalog entry UID by an

interaction with the catalog manager, and the UID is stored in the process descriptor.

A principal may have a default program registered with the Authentication Manager; if so, this program is executed at login time. If no program is specified, the standard command interpreter is assumed. The standard input and output for the executing process are directed to the principal's terminal device process.

13.4 Accessing a File

Each process descriptor contains (among other things) an entry for the UID of the current directory. This value is initialized at login to the principal's home directory, but can be modified during the course of the session. The current directory is inherited by a new program carrier process.

Suppose a client process wants to read the first 500 bytes of data in the primal file with the symbolic name :a:b:c. To do this, it would obtain the UID for the Primal File by means of.

```
Lookup(nullDirUID, ":a:b:c", true)
    -> abDirUID, abcCatEntUID, abcCatEntContents.
```

By convention, the UID for the null directory, nullDirUID, is used to specify the starting directory whenever a complete name is to be looked up. Next, it would read the file data by means of.

```
Read(abcCatEntContents.ObjectUID, 0, 500)
```

which would cause the primal file manager to send the first 500 bytes of data for the file.

These operations are made available by a single function call in the Process Support library.

```
ReadFileData(":a:b:c", 0, 500)
```

Now, assume that a process has a relative symbolic name for a file. The current directory UID is included in the request to the catalog to look up the file name. Using the general form of Invoke, the catalog manager is found based on the hint in the catalog entry UID. The catalog manager performs the lookup and returns the primal file UID associated with the symbolic name. The primal file UID is then used to find the file manager for this object, again using the hint which is part of the file UID to locate the manager.

13.5 Creating a File

A Cronus cluster may contain many hosts with file managers, each willing to store and retrieve file data at the request of other processes. The operation

```
Locate(CL_Primal_File)
```

can be invoked by a process to determine the set of accessible primal file managers.

One policy for the creation of files might be to try to create the file on the same host as the creating process if a local primal file manager responded. If this is not possible, a remote manager can be selected and asked to create the file. The primal files manager include status information, information in the responses, such the amount of unused disk storage available; a measure of the current I/O and processor load; or a restriction on the principal UIDs that may to create files through this manager. This information can be used to select a storage site for the file. The selection strategies are packaged in a library routines in the Process Support Library.

The file may need a symbolic catalog entry. The catalog entry operation is carried out by the catalog manager of the directory to which the file is being added.

Suppose that the client process wants to create a file and to give it the symbolic name a.b.c. Further suppose that a

directory named .a:b already exists.

First the client would use the

```
Create -> FileUID
```

operation to create a new primal file. The file would be empty. The client could write data into the file by means of:

```
Write(FileUID, BytePosition, Data)
```

or by bracketing the write(s) by

```
Open(FileUID, ReadWrite, Frozen)
```

and

```
Close(FileUID, RetainWrites)
```

operations

To catalog the file, the client first obtains the UID of the directory that will contain the catalog entry for the new name.

```
Lookup(nullDirUID, ".a:b", true)
    -> aDirUID, abCatEntUID, abCatEntContents
```

and then enters the new name.

```
Enter(abCatEntContents.ObjectUID, "c", FileUID)
    -> abcCatEntUID
```

If there were no directory :a.b or :a, then the client would first have to create both .a and .a.b. This could be done as follows. First the client would obtain the UID for the root directory. By convention the name of the root directory is .Root. The fact that the root directory is cataloged in itself represents the only violation of the tree structured property of the Cronus symbolic name space.

```
Lookup(nullDirUID, ".Root", true)
    -> rootDirUID,
       rootCatEntUID,
```

rootCatEntContents

Next, the client would create the directory :a:

```
CreateDir(rootDirUID, "a")
    -> aDirUID, aCatEntUID
```

and then, it would create the directory :a:b:

```
Create(aDirUID, "b") -> abDirUID, abCatEntUID.
```

At this point, the symbolic name .a.b.c can be established, as above, for the primal file.

The Process Support Library contains routines coupling the creation and naming of files, to avoid the situation where a failure produces a file which does not have a symbolic catalog entry and hence is not easily accessed. The operations are ordered such that the symbolic name is entered before the file is closed. If the process fails after the name is entered, the catalog entry may be deleted by explicit user commands, or by automatic recovery mechanisms.

13.6 Deleting a File

Suppose the name of the file to be deleted is >a>b>c. Deletion is accomplished by the following operations.

```
Lookup(nullDirUID, ":a:b:c", true)
    -> abDirUID, abcCatEntUID, abcCatEntContents

Delete(abcCatEntContents.ObjectUID)

Remove(abcCatEntUID)
```

If the primal file and catalog manager are coupled, the Delete operation could have the side effect of invoking the Remove operation.

13.7 Listing a Symbolic Catalog Directory

Suppose the name of the directory is :a:b:c. A utility program executes the following sequences of operations to print the desired file names.

```
InitScan(nullDirUID, ":a:b:c:*.*")
        -> abcScanState,
           xDirUID,
           xCatEntUID,
           xCatEntContents

repeat until abcScanState indicates end of scan
   [ if TypeOf(xCatEntContents.ObjectUID) = A_filetype
     then print xCatEntContents.SymbolicName;

     ScanDirectory(abcScanState)
           -> abcScanState,
              xDirUID,
              xCatEntUID,
              xCatEntContents,
]
```

13.8 Running a Program

Application programs are executed within program carrier objects. The creation of an application process has three steps: a program carrier is created, the program carrier is loaded with the program image, and the program carrier is started.

The program image will generally be obtained from a Cronus file, which may be anywhere within the Cronus file system. A routine, that combines these process creation steps process creation will be available in the PSL. This routine takes as one of its arguments the symbolic name of the program image file. The symbolic name is translated to the file UID by means of a symbolic catalog lookup, and the file UID is used to load the program image into a new program carrier object.

In a heterogeneous system, a particular program image can only be executed on certain processors. A VAX program image, for example, can only be executed on a VAX host. Some mechanism must exist to match the the program image to a processor capable of executing it.

Subtypes of program carriers are defined for each processor architecture for example, CT_VAX_Program_Carrier. These subtypes contribute no new operations to objects of type CT_Program_Carrier, but provide a means of locating a specific kind of processor. For example, the operation

```
Locate(CL_VAX_Program_Carrier)
```

will attempt to locate all program carrier managers on VAX hosts.

Executable files are subtypes of primal files with the type CT_Executable. The descriptor of a program image file contains the logical name of a program carrier subtype, e.g., CL_VAX_Program_Carrier. The file descriptor may also contain other information such as special host requirements. An operation on program carrier managers, Resource_Test, determines if a particular manager has the resources which are prerequisites to execution, the Create_Process routine can invoke this test whenever a process has special needs.

The actions carried out by the library routine can now be described in greater detail

1. The symbolic program name is translated to an executable file UID, by means of a symbolic catalog lookup.

2. The routine requests the file descriptor of the program image file, by invoking the Read_Descriptor on the file object.

3. The required program carrier type and any special requirements are determined from the file descriptor.

4. A Locate operation finds the Program Carrier Managers capable of executing this process, and a Resource_Test operation narrows the candidates further.

5. A Program Carrier Manager is selected according to some policy (26) and the operation Create_Program_Carrier is invoked on it; the UID of the new Program Carrier object is returned.

6. The Load_Program operation is invoked on the program carrier object.

7. When the load operation is complete, the routine receives a reply from the Program Carrier object, and then invokes Proceed on the Program Carrier to start it.

The Create_Program_Carrier operation takes as a implicit parameter the process descriptor of the creating process, which is inherited (with certain changes) by the new process.

A process descriptor contains some information which is maintained securely by the system (e.g., the process UID, and the UID of its principal) and an open-ended set of information inserted into the descriptor by the Change_Process_Descriptor operation. All of the open-ended information is inherited directly by the descendants of the process. Some of the system information is inherited (e.g., the principal is normally inherited) and some of it is not (e.g., the process UID of a descendant is unique to it). The system information defines the authority of the new process for access to information and resources.

The creating process may invoke Change_Process_Descriptor after but before starting, the program carrier to make changes in the descriptor.

---

(26) A reasonable policy might select the Program Carrier manager on the local host, if it is a candidate, and to select the most lightly loaded host (from information in the reply to Locate) if it is not. Many other policies are possible, and exploring the possibilities is an important area of future work.

13.9 Starting a Cronus Service

In this section we sketch a scenario which might be directed by a cluster control station, to startup, operate, and take down a Time Service instance on one host. It is indicative of the steps required to initiate and control an initial process load sequence. The steps required to bring up each host to the point assumed in this scenarios have been discussed in Section 12.

The Cronus Time Service has two main functions.

1. To respond to direct requests for the date and time, and for format conversions among the Cronus date and time formats.

2. To periodically multicast the date and time on a well-known VLN multicast channel.

Assume that host CVAX has joined the Cronus system, and the primal process manager is the only active Cronus process. The control station performs

```
InvokeOnHost("CVAX",
     CL_Primal_Process,
     <(CK_Operation_Name.CO_Service_List)>
)
```

and receives in reply a list of the services which could be created on CVAX, only the PPM is marked as active. The logical name CL_Time_Service is contained in the list. The control station then performs

```
InvokeOnHost("CVAX",
    CL_Primal_Process,
    <(CK_Operation_Name.CO_Create_Primal_Process)
     (CK_UID_Service_Name.CL_Time_Service)>)
```

The Time Service process is created and started, and the control station receives a reply containing CVAX_Time_Service_UID, the specific UID of the Time Service Primal Process. The Time Service begins its work, and if left undisturbed will periodically multicast the date and time forever. The control

station (or any other Cronus process) could request the current date and time by performing

```
InvokeOnHost("CVAX",
    CL_Time_Service,
    <(CK_Operation_Name,CO_Date_Time)>)
```

At some later time, it becomes necessary to temporarily inhibit the periodic multicasts of the Timer Service. The control station performs

```
InvokeOnHost("CVAX",
    CVAX_Time_Service_UID,
    <(CK_Operation_Name,CO_Change_Process_Descriptor),
     (CK_Modify,)
     (CK_IPCEnabled,"false")>)
```

After the control station receives the reply confirming this operation, it is known that all IPC to or from the Time Service has been inhibited. The Time Service process continues to exist, however, and is eventually restored to its normal function when the control station performs

```
InvokeOnHost("CVAX",
    CVAX_Time_Service_UID,
    <(CK_Operation_Name,CO_Change_Process_Descriptor),
     (CK_Modify,)
     (CK_IPCEnabled,"true")>)
```

Finally, perhaps in preparation for replacing the Time Service code with a new version, the control station does

```
InvokeOnHost("CVAX",
    CVAX_Time_Service_UID,
    <(CK_Operation_Name,CO_Destroy)>)
```

and the Time Service process is known to be destroyed when the reply arrives at the control station.

```
* Revision 1.1  83/06/06  10.39.32  bjw
* Initial revision
```

14 Primal System Hardware

The Advanced Development model of the Cronus distributed operating system will have three mainframe computers, four GCEs, and a gateway. The mainframe computers are two BBN C70s and a Digital Equipment Corporation VAX 11/750, the GCEs are Multibus computers with M68000 central processors, and the gateway is an DEC LSI-11 based computer.

The C70 computers are configured as general development machines. The first, C70-1, is the site of the majority of the development work since it supports both the C70 development tools and those of the GCEs. We will rent time on a second C70, C70-2, which will be used to exercise Cronus support for reliable redundant hosts, and to test scalability. Both C70s will run UNIX version 7 as released by BBN Computer Corporation and modified by the Cronus project.

The VAX 11/750 provides a VMS-based software development environment, as well as a mainframe of a distinctly different architecture. Its purpose in the ADM is to provide a limited integration host. Since it is a large well-supported mainframe, it will contain its own development environment, and we will also use it as a source of computer power for general tasks, both to off-load the C70, and to test real usage of the Cronus heterogeneous host environment. The VAX is configured to reflect its usage as a software development machine.

The Cronus system has four GCEs, configured for a variety of tasks. Since they are compatible machines, their configurations will vary over time, as we perform different experiments on the network, and as we make board substitutions to make one GCE perform functions of another which is temporarily out of service. The configuration table for the GCEs should be regarded as only a typical set of GCE configurations.

The Cronus gateway is implemented on an DEC LSI-11 computer. This would normally be a task for a GCE, however, standard internet gateways are currently implemented on LSI-11, and adoption of the LSI-11 gateway allows us to obtain an off-the-shelf implementation. The next generation of internet gateways is expected to be built on M68000 computers, and at that time we will probably move the gateway to a GCE.

C70-1	1 Mbyte main storage 2 80 Mbyte removable disk drives Magnetic Tape Drive, 800/1600 bpi, 125 ips (Cipher) Arpanet 1822 LHDH interface Ethernet interface (using Interlan protocol module)
C70-2	1/2 Mbyte main storage 2 160 Mbyte removable disk drives Arpanet 1822 LHDH interface Ethernet interface (using Interlan protocol module)
VAX 11-750	1 Mbyte main memory 1 160 Mbyte Winchester disk Magnetic tape drive, 1600 bpi, 40 ips MDI high speed synchronous serial interface 3COM Ethernet Interface VMS Operating System

Table 14.1 Software Development Hosts

GCE-1+2	Forward Technology M68000 processor with 256 Kbytes memory Micro-Memory 256 Kbyte memory board 80 Mbyte Winchester Disk Drive and SMD interface 3COM Ethernet Interface 8-slot Multibus backplane
GCE-3	Forward Technology M68000 processor with 256 Kbytes memory Micro-Memory 256 Kbyte memory board 8-line RS-232 serial interface 3COM Ethernet Interface 8-slot Multibus backplane
GCE-4	Forward Technology M68000 processor with 256 Kbytes memory Micro-Memory 256 Kbyte memory board 8-line RS-232 serial interface 300 lpm line printer 3COM Ethernet Interface 8-slot Multibus backplane

Table 14.2 Generic Computing Elements -- Typical Configurations

Gateway	LSI11/03 processor card
	64 Kbyte memory card
	DLV11J 4 line terminal card
	MRV11C ROM card (bootstrap)
	ACC 1822 interface with DMA
	Interlan NI2010 QBUS Ethernet controller
	BBN FNV11 Fibernet interface
	MDB backplane and power-supply.

Table 14.3 Gateway Configuration

15 Virtual Local Network

15.1 Purpose and Scope

The Cronus Virtual Local Network (VLN) provides interhost message transport in the Cronus Distributed Operating System. The VLN client interface is available on every Cronus host. Client processes can send and receive messages using specific, broadcast, or multicast addressing.

The VLN stands in place of a direct interface to the physical local network (PLN). This additional level of abstraction is defined to meet two major system objectives:

- o Compatibility. The VLN is compatible with the Internet Protocol (IP) and with higher-level protocols, such as the Transmission Control Protocol (TCP), based on IP.

- o Substitutability. Cronus software built above the VLN is dependent only upon the VLN interface and not its implementation. It is possible to substitute one physical local network for another provided that the VLN interface specification is satisfied.

This description assumes the reader is familiar with the concepts and terminology of the DARPA Internet Program; reference [NIC 1982] is a compilation of the important protocol specifications and other documents. Documents in [NIC 1982] of special significance here are [Postel 1981a] and [Postel 1981b].

The Advanced Development Model ADM will be connected to the ARPANET, and it is important that the ADM conform to the standard and conventions of the DARPA internet community. In addition, a large body of software has evolved, and continues to evolve, in the internet community. For example, protocol compatibility permits Cronus to assimilate existing software components providing electronic mail, remote terminal access, and file transfer.

The substitutability goal reflects the belief that different instances of Cronus will use different physical local networks. Substitution may be desirable for reasons of cost, performance, or other properties of the physical local network such as mechanical and electrical ruggedness.

Figure 1 shows the position of the VLN in the lowest layers of the Cronus protocol hierarchy. The VLN interface specification leaves programming details of the interface and host-dependent issues unspecified. The precise representation of the VLN data structures and operations will vary from machine to machine, but the functional capabilities of the interface are the same regardless of the host.

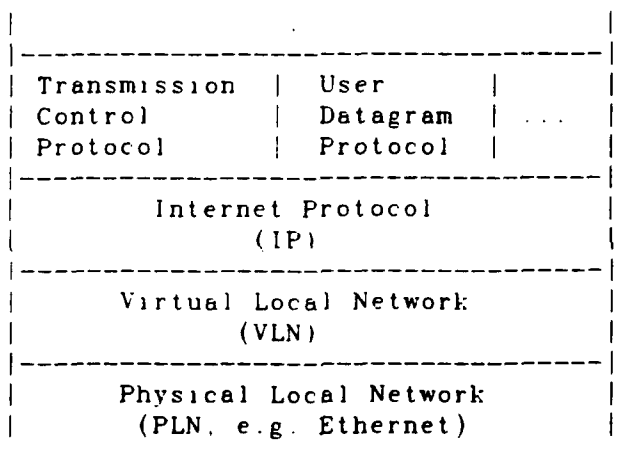


Figure 15.1 Cronus Protocol Layering

The VLN is completely compatible with the Internet Protocol as defined in [Postel 1981b]. No changes or extensions to IP are required to implement IP above the VLN.

15.2 The VLN-to-Client Interface

The VLN layer provides a datagram transport service among hosts in a Cronus cluster, and between these hosts and other hosts in the DARPA internet. The hosts belonging to a cluster are attached to the same physical local network. Communication with hosts outside the cluster is achieved through internet gateways, shown in Figure 2, connected to the cluster. The VLN routes datagrams to a gateway if they are addressed to hosts outside the cluster, and delivers incoming datagrams to the appropriate VLN host. A VLN is a network in the internet, and thus has an internet network number(27).

(27). The network numbers for the PLN and VLN may be the same or different. If the numbers are different, the gateways are somewhat more complex. Either approach is consistent with the internet model.

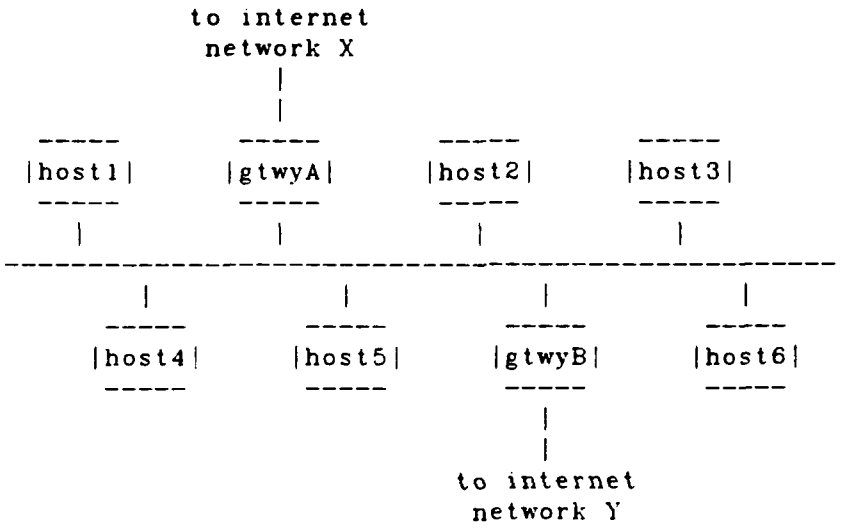


Figure 15.2 A Virtual Local Network Cluster

The VLN interface will have one client process on each host, normally the host's IP implementation. The VLN performs no multiplexing/ demultiplexing function.

The structure of messages which pass through the VLN is identical to the structure of internet datagrams. The VLN definition assumes that there is a well-defined representation for internet datagrams on any host supporting the VLN interface. The argument name "Datagram" in the VLN operation definitions below refers to this well-defined but host-dependent datagram representation.

The VLN guarantees that a datagram of 576 or fewer octets can be transferred between any two VLN clients. Although larger datagrams may be transferred between some client pairs, clients should avoid sending datagrams exceeding 576 octets unless there

is clear need to do so. The sender must be certain that all hosts involved can process the oversized datagrams.

The internal representation of an VLN datagram is not included in the specification, and may be chosen for implementation convenience or efficiency.

Although the structure of internet and VLN datagrams is identical, the VLN-to-client interface places its own interpretation on internet header fields, and differs from the IP-to-client interface in significant respects.

1. The VLN layer uses only the Source Address, Destination Address, Total Length, and Header Checksum fields in the internet datagram; other fields are accurately transmitted from the sending to the receiving client.

2. Internet datagram fragmentation and reassembly is not performed in the VLN layer, nor does the VLN layer implement any aspect of internet datagram option processing.

3. At the VLN interface, a special interpretation is placed upon the Destination Address in the internet header, which allows VLN broadcast and multicast addresses to be encoded in the internet address structure.

4. With high probability, duplicate delivery of datagrams sent between hosts on the same VLN does not occur.

5. Between two VLN clients S and R in the same Cronus cluster, the sequence of datagrams received by R is a subsequence of the sequence sent by S to R; a stronger sequencing property holds for broadcast and multicast addressing.

In the DARPA internet, an internet address is defined to be a 32-bit quantity that is partitioned into two fields, a network number and a local address. VLN addresses share this basic structure, but it attaches special meaning to the local address field of a VLN address.

Each network is assigned a class (A, B, or C), and a network number. The partitioning of the 32-bit internet address into

network number and local address fields as a function of the class of the network is shown in Table 1.

	Width of Network Number	Width of Local Address
Class A	7 bits	24 bits
Class B	14 bits	16 bits
Class C	21 bits	8 bits

Table 15.1 Internet Address Formats

The bits not included in the network number or local address fields encode the network class, e.g., a 3 bit prefix of 110 designates a class C address (see [Postel 1981a]).

The interpretation of the local address field is the responsibility of the network. For example, in the ARPANET the local address refers to a specific physical host. VLN addresses, in contrast, may refer to all hosts (broadcast) or groups of hosts (multicast) in a Cronus cluster, as well as specific hosts inside or outside of the cluster. Specific, broadcast, and multicast addresses are all encoded in the VLN local address field (28). The meaning of the local address field of a VLN address is defined in Table 2.

(28). The ability of hosts outside a Cronus cluster to transmit datagrams with VLN broadcast or multicast destination addresses into the cluster may be restricted by the cluster gateway(s), for reasons of system security.

Address Modes	VLN Local Address Values
Specific Host	0 to 1,023
Multicast	1,024 to 65,534
Broadcast	65,535

Table 15.2 VLN Local Address Modes

In order to represent the full range of specific, broadcast, and multicast addresses in the local address field, a VLN network should be either class A or class B.

The VLN does not attempt to guarantee reliable delivery of datagrams, nor does it provide negative acknowledgements of damaged or discarded datagrams. It does guarantee that received datagrams are accurate representations of transmitted datagrams.

The VLN guarantees that datagrams will not replicate during transmission, so each intended receiver, a given datagram given to the VLN by higher levels is received once or not at all(29).

Between two VLN clients S and R in the same cluster, the sequence of datagrams received by R is a subsequence of the sequence sent by S to R, that is datagrams are received in order, possibly with omissions. A stronger sequencing property holds for broadcast and multicast transmissions. If receivers R1 and R2 both receive broadcast or multicast datagrams D1 and D2, either they both receive D1 before D2, or they both receive D2 before D1.

While a VLN could be implemented on a long-haul or virtual-

(29). A protocol operating above the VLN layer (e.g., TCP) may employ a retransmission strategy; the VLN layer does nothing to filter duplicates arising in this way.

circuit-oriented PLN, these networks are generally ill-suited to the task. The ARPANET, for example, does not support broadcast or multicast addressing modes, nor does it provide the VLN sequencing guarantees. If the ARPANET were the base for a VLN implementation, broadcast and multicast would have to be constructed from specific addressing, and a network-wide synchronization mechanism would be required to implement the guarantees. Although the compatibility and substitutability benefits might still be achieved, the implementation would be costly, and performance poor.

A good implementation base for a Cronus VLN would be a high-bandwidth local network with all or most of these characteristics.

1. The ability to encapsulate a VLN datagram in a single PLN datagram.

2. An efficient broadcast addressing mode.

3. Natural resistance to datagram replication during transmission.

4. Sequencing guarantees like those of the VLN interface.

5. A strong error-detecting code (datagram checksum).

Good candidates include Ethernet, the Flexible Intraconnect, and Pronet, among others.

15.3 A VLN Implementation Based on Ethernet

The Ethernet local network specification is the result of a collaborative effort by Digital Equipment Corp., Intel Corp., and Xerox Corp. The Version 1.0 specification [DEC 1980] was released in September 1980. Useful background information on the Ethernet internet model is supplied in [Dalal 1981].

The addresses of specific Ethernet hosts are arbitrary 48-bit quantities, not under the control of the DOS. The VLN implementation must map VLN addresses to specific Ethernet addresses. The mapping can not be maintained manually in each

VLN host, because manual procedures are too cumbersome and error-prone for a local network with many hosts, each of which may join and leave the network frequently. A protocol is described below which allows a host to construct the mapping dynamically, beginning only with knowledge of its own VLN and Ethernet host addresses.

An internet datagram is encapsulated in an Ethernet frame by placing the internet datagram in the Ethernet frame data field, and setting the Ethernet type field to "DoD IP", as shown in Figure 3.

The Ethernet octet ordering is required to be consistent with the IP octet ordering. If IP(i) and IP(j) are internet datagram octets and i<j, and EF(k) and EF(l) are the Ethernet frame octets which represent IP(i) and IP(j) once encapsulated, then k<l. Bit orderings within octets must also be consistent.

Each VLN component maintains a virtual-to-physical address map (the VPMap) which translates a 32-bit specific VLN host address to a 48-bit Ethernet address. The VPMap data structure and the operations on it will implemented using hashing techniques.

Each host controller has an Ethernet host address (EHA) to which it responds. The EHA is determined by Xerox and the controller manufacturer. In addition, the VLN assigns a multicast-host address (MHA) to each host. This multicast address is constructed from the local host portion of the internet address.

When the VLN client sends a datagram to a specific host, the local VLN component encapsulates it and transmits it without delay. The Source Address in the Ethernet frame is the EHA of the sending host. The Ethernet Destination Address is formed from the destination VLN address in the datagram, and is either:

- o the EHA of the destination host, if the sending host knows it, or
- o the MHA formed from the host number in the destination VLN address, as described above, if the sending host does not know the EHA coresponding to the host number.

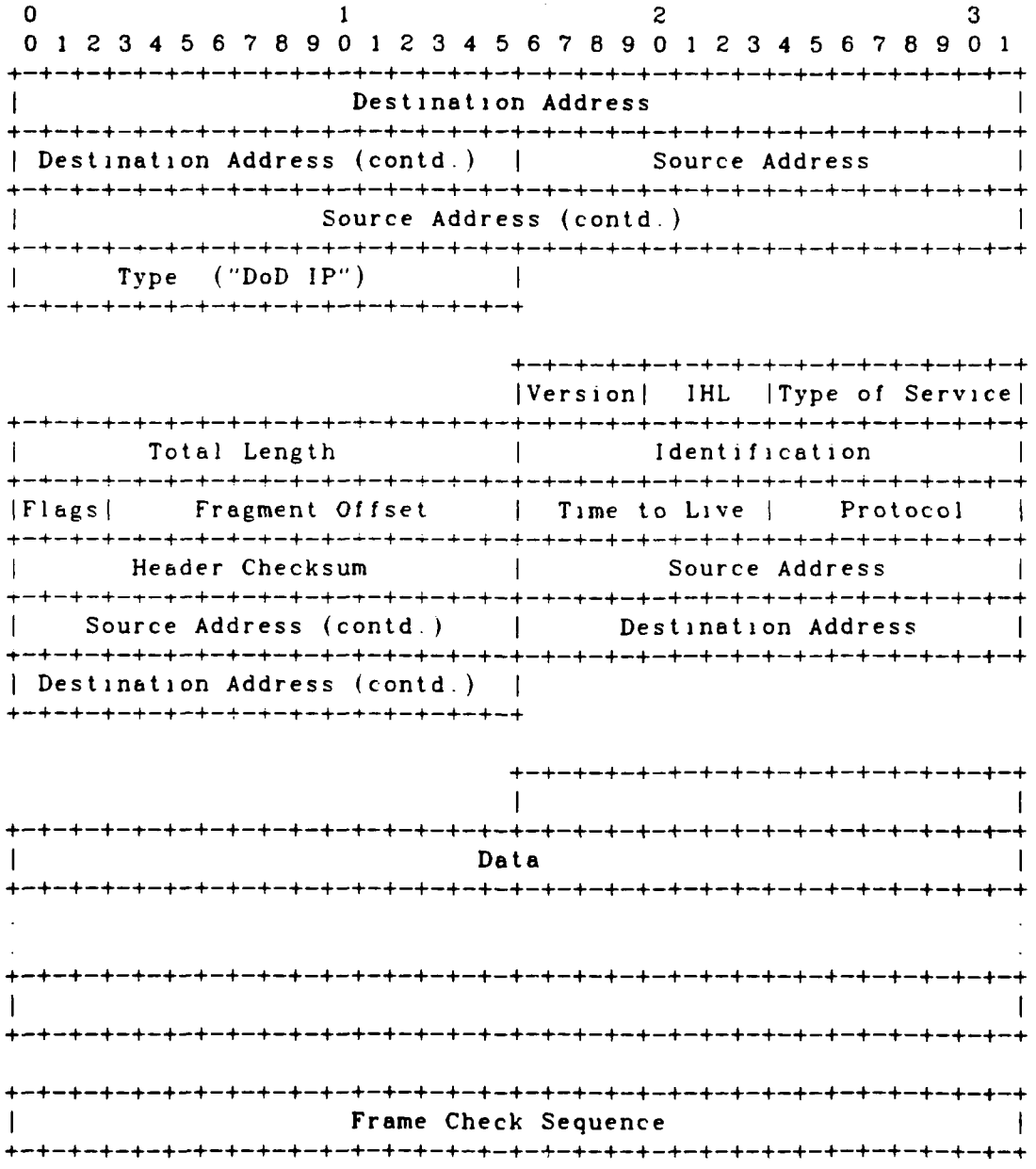


Table 15.3 An Encapsulated Internet Datagram

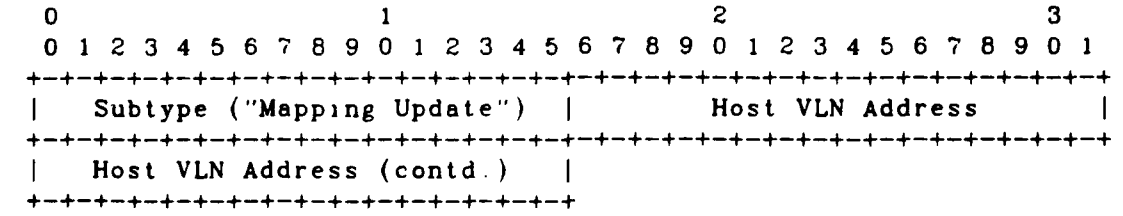
When a VLN component receives an Ethernet frame with type "DoD IP", it decapsulates the internet datagram and delivers it to its client. If the frame was addressed to the EHA of the receiving host, no further action is taken. If the frame was addressed to the MHA of the receiving host, the VLN component broadcasts an update for the VPMaps of the other hosts. The other hosts can thenuse the EHA of this host for future traffic. If the MHA is represented as a sequence of octets in hexadecimal, it has the form.

```
A  B  C  D  E  F

09-00-08-00-hh-hh
```

A is the first octet transmitted, and F the last. The two octets E and F contain the host local address.

```
   E         F

000000hh  hhhhhhhh
      ↑         ↑
    MSB       LSB
```

The type field of the Ethernet frame containing the update is "Cronus VLN", and the format of the data octets in the frame is:



When a local VLN component receives an Ethernet frame with type "Cronus VLN" and subtype "Mapping Update", it performs a

StoreVPPair operation using the Ethernet Source Address field and the host VLN address sent as frame data.

A VLN datagram will be transmitted in broadcast mode if the specifies the VLN broadcast address (local address = 65,535, decimal) as the destination. The receiving VLN component merely decapsulates and delivers the VLN datagram.

The implementation of multicast addressing is more complex. Each host defines the number of multicast addresses which can be simultaneously "attended" (listened to). This number is a function of the particular Ethernet controller hardware and of the resources that the host dedicates to multicast processing. The VLN protocol permits a host to attend any number of multicast addresses, from 0 to 64,511 (the entire VLN multicast address space), independent of the controller in use.

It is possible to implement the VLN multicast mode using only the Ethernet broadcast mechanism. Every VLN host would receive and process every VLN multicast, discarding uninteresting datagrams. More efficient operation is possible if some Ethernet multicast addresses are used, and if the Ethernet controller has multicast recognition which automatically discard misaddressed frames.

There is no standard for multicast recognition. The 3COM Model 3C400 controller performs no multicast address recognition. It passes all multicast frames to the host for further processing. The Intel Model iSBC 550 controller permits the host to register a maximum of 8 multicast addresses with the controller, and the Interlan Model NM10 controller permits a maximum of 63 registered addresses.

A VLN-wide constant, Multicast_Registered, is equal to the smallest number of Ethernet multicast addresses that can be simultaneously attended by all hosts in the VLN. A network composed of hosts with the Intel and Interlan controllers mentioned above, for example, would have Multicast_Registered equal to 7 (30), a network composed only of hosts with 3COM Model 3C400 controllers would have Multicast_Registered equal to

(30). Multi_Registered is 7, rather than 8, because one multicast slot in the controller is reserved for the host's MHA.

64,511, since the controller itself does not restrict the number of Ethernet multicast addresses to which a host may attend (31).

A mapping is defined which translates the VLN multicast address to an Ethernet multicast address. The first Multicast_Registered VLN multicast addresses are assumed to be attended by each host. The local address portion of the internet address of a VLN multicast channel is a decimal integer M in the range 1,024 to 65,534.

1. (M - 1,023) <= Multicast_Registered. In this case, the Ethernet multicast address is

   09-00-08-00-mm-mm

2. (M - 1,023) > Multicast_Registered. The Ethernet broadcast address is used. A VLN component which attends VLN multicast addresses in this range must receive all broadcast frames, and select those with VLN destination address corresponding to the attended multicast address.

Delivered datagrams are accurate copies of transmitted datagrams because VLN components do not deliver datagrams with invalid Frame Check Sequences. A 32-bit CRC error-detecting code is applied to Ethernet frames.

Datagram duplication does not occur because the VLN layer does not perform retransmissions, the primary source of duplicates in other networks. Ethernet controllers do perform retransmission as a result of collisions on the channel, but the collision enforcement mechanism or "jam" assures that no controller receives a valid frame if a collision occurs.

The sequencing guarantees hold because mutually exclusive access to the transmission medium defines a total ordering on Ethernet transmissions, and because a VLN component buffers all datagrams in FIFO order.

---

(31). For the Cronus Advanced Development Model, Multicast_Registered is currently defined to be 60.

15.4 VLN Operations

There are seven functions defined at the VLN interface. An implementation of the VLN interface has wide latitude in the presentation of these operations to the client; for example, the functions may or may not return error codes.

The functions are to occur synchronously or asynchronously with respect to the client's computation. We expect that the ResetVLNInterface, MyVLNAddress, SendVLNDatagram, PurgeMAddresses, AttendMAddress, and IgnoreMAddress operations will be synchronous with respect to the client. ReceiveVLNDatagram will usually be asynchronous, that is, the client initiates the operation, continues to compute, and at some later time is notified that a datagram is available.

ResetVLNInterface()

> The VLN for this host is reset. For the Ethernet implementation, the operation ClearVPMap is performed, and a frame of type "Cronus VLN" and subtype "Mapping Update" is broadcast. This operation does not affect the set of attended VLN multicast addresses.

MyVLNAddress()

> Returns the VLN address of this host.

SendVLNDatagram(Datagram)

> When this operation completes, the VLN layer has copied the Datagram. The transmitting process cannot assume that the message has been delivered when SendVLNDatagram completes.

ReceiveVLNDatagram(Datagram)

> When this operation completes, Datagram is a representation of a VLN datagram which has not previously received.

PurgeMAddresses()

> When this operation completes, no VLN multicast addresses

are registered with the local VLN component.

AttendMAddress(MAddress)

If this operation returns True then MAddress, which must be a VLN multicast address, is registered as an alias for this host, and messages addressed to MAddress by VLN clients will be delivered to the client on this host.

IgnoreMAddress(MAddress)

When this operation completes, MAddress is not registered as a multicast address for the client on this host.

Whenever a Cronus host comes up, ResetVLNInterface and PurgeMAddresses are performed on the VLN. A VLN component may depend upon state information obtained dynamically from other hosts, and there is a possibility that incorrect information might enter a component's state tables. A cautious VLN client could call ResetVLNInterface periodically to force the VLN component to reconstruct the tables.

A VLN component will limit the number of multicast addresses to which it will simultaneously attend; if the client attempts to register more addresses than this, AttendMAddress will return False with no other effect.

The VLN layer does not guarantee buffering for datagrams at either the sending or receiving host(s). It does guarantee that a SendVLNDatagram function performed by a VLN client will eventually complete; this implies that datagrams may be lost if buffering is insufficient and receiving clients are too slow.

16 Generic Computing Element Operating System

One of the more important Cronus hardware components is the Generic Computing Element (GCE). Prior to its introduction to the Cronus DOS project, CMOS was under development at BBN as a real-time operating system for several types of communication processors, such as gateways and network terminal concentrators. In addition, a support environment for building and debugging CMOS applications is available under UNIX. CMOS provides the following basic operating system features:

- o multiple processes
- o interprocess communication/coordination
- o asynchronous I/O
- o memory allocation
- o system clock management

CMOS is an open operating system, that is, no distinct division exists between the operating system and the application program. The operating system is a collection of library routines that can be easily extended by adding new routines and can be reduced by excluding unneeded routines. The programmer can directly access lower-level interfaces.

CMOS is a portable operating system. The use of the high-level language C is the principal factor in CMOS portability. Small size and simplicity are other important factors. The design minimizes the amount of machine-dependent code and segregates it. The I/O system design allows for easy replacement of device-dependent modules.

The debugging environment is provided by XMD, a display oriented debugger based on the PEN editor. All of the features of the editor are available to the user in addition to the debugger specific commands. PEN is a multi-window editor with capabilities for manipulating multiple files and edit buffers. XMD displays a special configuration of windows that are appropriate to debugging. This configuration consists of a source window, a register display window, a breakpoint window, and a window for displaying variables.

A low-level debugger is resident in the target processor to interpret and execute commands sent to it over the communication path, currently a terminal line to the C70 UNIX host processor where XMD is running.

Access to networks will be provided to CMOS applications from three levels. At the highest level, the user can open a TCP stream. The first application at this level will be Telnet and terminal concentration software. At the next level, there is an internet datagram service. This will be used to implement inter-process communication between hosts, as well as other standard internet protocols. The lowest level is the Ethernet local network interface.

The communication module in XMD will be changed to use the Ethernet instead of a terminal line, increasing its flexibility and usefulness. Downloading will be possible over the network, plus it will be easier to debug multiple GCEs from one site.

The internal device structure was changed was to give the I/O system more flexibility in dealing with the number of possible relationships between hardware devices and the interrupts generated by those devices. Without this change, the capability of writing simple device drivers for CMOS is compromised.

A name service capability was added for the run-time binding of string names to processes and devices. The name space is hierarchical and there is a notion of absolute and relative pathnames. In the presence of some form of mass storage, the names can be made non-volatile.

REFERENCES

[BBN 5041]
"Cronus, a distributed operating system: functional definition and system concept," M. D. Hoffman, W. I. MacGregor, R. E. Schantz, & R. H. Thomas, Technical Report #5041, Bolt Beranek and Newman Inc., June 1982.

[BBN 5086]
"Cronus, A Distributed Operating System. Interim Technical Report No. 1," R. Schantz, E. Burke, S. Geyer, M. Hoffman, A. Lake, K. Pogran, D. Tappan, R. Thomas, S. Toner, and W. MacGregor, Technical Report #5086, Bolt Beranek and Newman Inc., July 1982.

[BBN 5260]
Part A of [BBN 5261]

[BBN 5261]
"Cronus, A Distributed Operating System. Interim Technical Report No. 2," R. Schantz, B. Woznick, G. Bono, E. Burke, S. Geyer, M. Hoffman, W. MacGregor, R. Sands, R. Thomas, and S. Toner Technical Report #5261, Bolt Beranek and Newman Inc., February 1983.

[Dalal 1981]
"48-bit absolute internet and Ethernet host numbers," Yogen K. Dalal and Robert S. Printis, Proc. of the 7th Data Communications Symposium, October 1981.

[DEC 1980]
"The Ethernet: a local area network, data link layer and physical layer specifications," Digital Equipment Corp., Intel Corp., and Xerox Corp., Version 1.0, September 1980.

[Goldberg 1983]
"Smalltalk-80. The Language and Its Implementation", Adele Goldberg and David Robson, Addison-Wesley, Reading Ma, 1983.

[Herlihy 1982]
"A Value Transmission Method for Abstract Data Types", M. Herlihy

and B. Liskov, ACM Transactions on Programming Languages and Systems Volume 4 (4) 527, October 1982.

[Jones 1978]
"The Object Model: A Tool for Structuring Software," A. K. Jones, in "Operating Systems. An Advanced Course," R. Bayer, R. M. Graham, and G. Seegmuller, eds., Springer-Verlag, Heilelberg, 1978.

[Liskov 1977]
"An Introduction to Formal Specifications of Data Abstractions," Barbara Liskov and Stephen Ziles, in "Current Trends in Programming Methodology", Vol 1, Raymond T. Yeh, ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[NIC 1982]
"Internet protocol transition workbook," Network Information Center, SRI International, Menlo Park, California, March 1982.

[Parker 1983]
"Detection of Mutual Inconsistency in Distributed Systems," D. S. Parker, Jr, et al, IEEE Transactions on Software Engineering, Volume SE-9 (3) 240, May 1983.

[Postel 1981a]
"Assigned numbers," Jon Postel, RFC 790, USC/Information Sciences Institute, September 1981.

[Postel 1981b]
"Internet Protocol - DARPA internet program protocol specification," Jon Postel, ed., RFC 791, USC/Information Sciences Institute, September 1981.

[Rentsch 1982]
"Object oriented programming," T. Rentsch, SIGPLAN Notices, September 1982, pp. 51-57.

[Robinson 1977]
"A Formal Methodology for the Design of Operating System Software," Lawrence Robinson, Karl N. Levitt, Peter G. Neumann, and Ashok R Saxena, in "Current Trends in Programming Methodology", Vol 1, Raymond T. Yeh, ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[Weinreb 1981]

"Lisp Machine Manual", Daniel Weinreb and David Moon, Massachusetts Institute of Technology, Cambridge Ma., 4th ed., 1981, p. 279f.

INDEX