(4)

AD-A199 892

RADC-TR-88-132, Vol IV (of four)
Final Technical Report
June 1988

CRONUS, A DISTRIBUTED OPERATING SYSTEM: CRONUS DOS Implementation

BBN Laboratories Incorporated

R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean,
R. Gurwitz, K. Lam, K. Lebowitz, S. Lipsôn, P. Neves and R. Sands

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, 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.

RADC-TR-88-132, Volume IV (of four) has been reviewed and is approved for publication.

APPROVED:

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

JOH'' A. RITZ
Directorate of Plans and 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 notices on a specific doucment require that it be returned.

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

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
Report No. 5183

5. MONITORING ORGANIZATION REPORT NUMBER(S)
RADC-TR-88-132, Volume IV (of four)

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
F30602-84-C-0171

8c. ADDRESS (City, State, and ZIP Code)
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
63728F	2530	01	26

11. TITLE (Include Security Classification)
CRONUS, A DISTRIBUTED OPERATING SYSTEM: CRONUS DOS Implementation

12. PERSONAL AUTHOR(S) R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lam, K. Lebowitz, S. Lipson, P. Neves and R. Sands

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM Oct 84 TO Jan 86

14. DATE OF REPORT (Year, Month, Day)
June 1988

15. PAGE COUNT
64

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	07	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Distributed Operating System, Heterogeneous Distributed System, Interoperability, System Monitoring & Control, Survivable Application

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This is the final report for the second contract phase for development of the Cronus Project. Cronus is the name given to the distributed operating system (DOS) and system architecture for distributed application development environment being designed and implemented by BBN Laboratories for the Air Force Rome Air Development Center (RADC). The project was begun in 1981. The Cronus distributed operating system is intended to promote resource sharing among interconnected computer systems and manage the collection of resources which are shared. Its major purpose is to provide a coherent and integrated system based on clusters of interconnected heterogeneous computers to support the development and use of distributed applications. Distributed applications range from simple programs that merely require convenient reference to remote data, to collections of complex subsystems tailored to take advantage of a distributed architecture. One of the main contributions of Cronus is a unifying architecture and model for developing these distributed applications, as well as support for a number of system-provided functions which are common to many applications. (Over)

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)

UNCLASSIFIED

Block 19 (Cont'd)

This report consists of four volumes:

- Vol I - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Revised System/Subsystem Specification
- Vol II - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Functional Definition and System Concept
- Vol III - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Interim Technical Report No. 5
- Vol IV - CRONUS, A DISTRIBUTED OPERATING SYSTEM: CRONUS DOS Implementation

UNCLASSIFIED

Table of Contents

FIGURES

1. Introduction

This is the final report for the second contract phase for development of the Cronus Project. Cronus is the name given to the distributed operating system (DOS) and system architecture for distributed application development environment being designed and implemented by BBN Laboratories for the Air Force Rome Air Development Center (RADC). The project was begun in 1981. The Cronus distributed operating system is intended to promote resource sharing among interconnected computer systems and manage the collection of resources which are shared. Its major purpose is to provide a coherent and integrated system based on clusters of interconnected heterogeneous computers to support the development and use of distributed applications. Distributed applications range from simple programs that merely require convenient reference to remote data, to collections of complex subsystems tailored to take advantage of a distributed architecture. One of the main contributions of Cronus is a unifying architecture and model for developing these distributed applications, as well as support for a number of system provided functions which are common to many applications.

This work is a continuation of research and development performed under the previous *DOS Design/Implementation* effort funded by RADC. For a description of previous Cronus development, see *CRONUS, A Distributed Operating System: Phase 1 Final Report**. During that initial phase, the functional description, system design and initial system implementation were completed.

To satisfy the need for ongoing test and evaluation of the system, particularly its suitability to application development in addition to the system development, we have also been performing an adjunct C^2 Internet Experiment project. Under the C^2 Internet Experiment effort, we have been building a prototype distributed command and control application which emulates many of the facilities of real C^2 systems. The ongoing evaluation of the Cronus mechanisms and tools contributed by this project has become an integral part of the development process. For a description of the C^2 Internet project, see *C^2 Internet Expriment: Final Report***.

2. Project Overview

This report covers Cronus development for the period from October 1984 to January 1986. The objective of this phase was to extend the Cronus Distributed Operating System implementation, completing the basic functionality for supporting distributed system demonstration software; to extend the testbed environment with additional hosts and tools to support the development and evaluation of Air Force applications; and to begin to establish a second testbed cluster on-site at RADC. The overall function of the DOS is to integrate the various data processing subsystems into a coherent, responsive and reliable system which supports development of distributed command and control applications. The development work for this contract is broken down into the following areas:

***CRONUS, A Distributed Operating System: Phase 1 Final Report**, *R. Schantz, et al.* BBN Report No. 5885. BBN Laboratories. Incorporated, January 1985.

****C^2 Internet Experiment: Final Report**, *J. Berets, et al.* BBN Laboratories, Incorporated, March 1985.

Area	SOW Item
VAX-UNIX Integration	4.1.1.2.1
SUN Workstation Integration and Use	4.1.1.2.2
Resource Management	4.1.2
Survivability	4.1.3
Reconfiguration Support	4.1.3.2
Tool Integration	4.1.4.1
Application Development Support	4.1.4.2
RADC Cluster Support	4.1.5
Cluster Maintenance	4.1.6

In addition to the development work, a description of how new hosts and their resources are integrated into a Cronus cluster was written and is included in this report.

Development of this system has followed a clearly defined, experimental approach. When extending the capabilities of the system we first identify suitable mechanisms for supporting the new facilities. Then, after some additional consideration, an initial implementation is produced for use by system components, such as the file or catalog manager. In cases where a mechanism might be supported in more than one place, we make our choice to satisfy a compromise between availability for other clients and facilities, implementation speed, and expected performance. This initial implementation is used to evaluate the suitability of the mechanisms to the problem it was intended to solve, and the mechanism will then be revised in response to criticism. When appropriate, the mechanisms will be introduced into manager and application development tools, and appropriate changes will be made. As the mechanisms are introduced for use at the application level, more attention is paid to interfaces and the model by which the user will understand the system's behavior; these factors are also considered in the early stages, but are not as important as having an initial version available for use and evaluation within the system.

3. Integration of New System Hardware

Under the previous Cronus development effort we established an initial demonstration environment consisting of utility hosts and Generic Computing Elements (GCEs). Our demonstration environment included two types of utility hosts for development support and application program execution: BBNCC C70 running UNIX and DEC VAX 11/750 running VMS. Most of our development activities were centered on C70 Unix because of the ease of developing new software afforded by the UNIX environment, and its rich set of development tools. The GCEs are small dedicated-function computers of a single architecture but varying configurations. In our demonstration environment we had several Motorola 68000 Multibus microprocessor systems running the CMOS operating system. They provided specific Cronus services, such as file management and terminal access points.

Under the present effort, we have expanded the development environment in three ways. First, we have added support for the SUN Workstation. The SUN Workstation represents a new class of Cronus host, oriented toward providing access dedicated to a single user. This type of system was included in the original hardware architecture design for Cronus but was not supported previously. Second, we have expanded the set of utility hosts to include VAX UNIX systems. The VAX-UNIX represents an evolution of the existing Cronus timesharing and peripheral support to a new, more advanced hardware base. And third, we have added limited support for access to resources on other local area networks. Currently this allows clients on our Cronus cluster to gain remote access to devices, such as line printers, located on other networks but accessible through the DARPA Internet. This represents the beginning of shared access among resources on remote clusters connected by networks with varying performance capabilities integrated within the Cronus system model.

3.1. VAX-UNIX

The VAX-UNIX system serves in the role of an application development host. Existing editors, compilers, libraries and other Unix tools form the foundation for development support; enhancements to these tools through *trap-libraries*, allow these tools to manipulate Cronus files and directories. Cronus tools running on these hosts extend the support to simplify the development of managers and applications. Further, because Unix is a timesharing system, the development resources provided by the host may be shared among many users, including many who are not involved in Cronus development. Finally, as a service host, the Unix systems provide access to line printers and file storage space, which can be accessed remotely through Cronus from any Cronus host in the cluster.

The VAX-UNIX systems support the Cronus operation switch, all system managers, including the file and catalog manager, several C^2 Internet application managers, all the application development tools, and all Cronus user commands. To speed development of Unix based utilities for accessing Cronus files, we have modified the standard C compiler libraries so that file I/O routines will invoke the appropriate Cronus operations whenever a Cronus file name is given. Simply recompiling many UNIX file utilities, such as *cp*, *cat*, *grep*, and *diff*, and the text editors *emacs* and *vi*, now produces versions that access both Unix and Cronus files. In some cases, minor modifications were required to the source programs.

We have implemented Cronus for VAX-UNIX to run on both the VAX 11/750 and 11/785. The hardware base for these implementations are currently owned and operated by the BBN Computer Systems Division to supply timesharing support for the company. The larger of the machines, the 11/785, typically supports 40-50 users. Cronus applications run concurrently with non-Cronus timesharing workload on these hosts.

The VAX-UNIX system serves to replace the C70 as a hardware base for future DOS and related application development. The VAX family of computers is widely accepted, with a large installed hardware base, which increases the likelihood of finding existing machines to integrate into Cronus. The VAX supports hardware architecture advances beyond the C70, including a large virtual address space managed under the Berkeley 4.2BSD release of UNIX. In addition to virtual memory support, the 4.2BSD provides many new features and languages, improved interprocess communication and I/O facilities, and better overall performance.

3.2. SUN Workstation Integration and Use

The SUN Workstation provides most of the facilities of an application development host. In addition, workstations provide powerful computation and high performance graphic capabilities dedicated to a single-user. These capabilities make feasible man-machine interfaces of significantly higher quality than those possible on time-shared mainframe computers communicating with terminals over slow, bit-serial links. To experiment with these additional capabilities, particularly the use of graphics, we have implemented a prototype monitoring and control interface on the SUN Workstation. These capabilities are also exploited in application interfaces for the related C^2 Internet experiment project.

The SUN Workstation is representative of the trend toward more powerful, single-user graphics workstations; there are others, including those produced by Digital, MassComp and Apollo, and those that can be expected from IBM and others in the future. The SUN Workstation is a Motorola 68000 Multibus system based on the SUN microprocessor board developed at Stanford University. It includes a high-resolution, bitmap, raster graphics display, with keyboard and mouse input devices, and a window based user interface. The system supports a version of Berkely 4.2BSD Unix, essentially the same as the VAX-UNIX described above, with virtual memory.

We have installed two Sun Model 120 Workstations, each with a 130 megabyte Winchester disk drive and 2 megabytes of primary memory. These systems offer enough power for use as workstations or for use as utility hosts for program development by 2-3 users performing typical development tasks. The workstations support the Cronus operation switch, all managers, including the file and catalog manager, all the application development tools and all Cronus user commands. The sources for these Cronus programs are essentially identical to the sources used for the VAX-UNIX system.

The workstation also supports the console interface for the Monitoring and Control System (**MCS**), a distributed application that monitors system status and behavior. The monitoring components include managers, running on other hosts, that monitor host availability, managers that log events and errors, and a manager that maintains configuration information used by the Cronus kernels. The manager components monitor, collect and maintain copies of the status and event data; the console interface is used by an operator to examine the data and initiate changes to component status and resource management parameters.

Two kinds of data are displayed on the workstation, event reports and status information. The event reports originate from managers or client programs when they submit an *event report* to the *event collector*, as when a Cronus host manager announces that a manager has crashed and is being automatically restarted. These reports are recorded by the collector and a copy is forwarded to the program managing the event report window on the workstation. When the report is displayed, it is expected that the operator will use the MCS console to correct any problem indicated by the report. The event display program requires no special terminal capabilites and therefore, can be run on a conventional terminal.

Status data is currently both collected and displayed by the console interface program. The collector portion gathers the status information by polling the managers for information about the objects they manage. The status collector can be set to monitor particular parameters and alert the operator through the event reporting facilities if a specified threshold is violated. The display portion permits the composition of interactive graphical diagrams, or *views*. These diagrams resemble control panels that can be connected to data sources and sinks and used to graphically control and display the state information. Using this system we have produced views that summarize cluster host status, the status of each of the services, and the status of the managers for each service.

3.3. Remote Resource Access

The processing nodes of the initial Cronus cluster were all connected via a single Ethernet Local Area Network (LAN). While the architecture of Cronus requires only a minimal set of interhost communication facilities, described as the *virtual local network*, the implementation of Cronus exploits special capabilities of true local area networks, such as broadcasting, and exploits the high bandwidth of these networks. To begin extending the range of resource access available to clients in a Cronus cluster, we have installed Cronus on a few systems accessible through a gateway connecting the Cronus Ethernet to the DARPA Internet, which in turn provides the connection either to the target system or its local area network.

Cronus was initially designed and implemented for an extended cluster internet environment, even though the initial components were, in fact, on a single local area network. To facilitate the later introduction of true internet capabilities, the Cronus message passing design is based on the standard Internet datagram protocol, IP. Thus, from the outset, operation invocations could traverse many networks, relying only on the IP message base. For primal objects, the UID for each object identifies the host where the object resides. Hence locating primal objects requires no additional work. The location of replicated and migratable objects must be discovered before operations can be invoked on such objects. We have used the broadcast services provided by the local area network as a basis for dynamically locating resources (objects) anywhere in our cluster. To extend this to multiple local area networks, we have added a broadcast repeater, which propagates broadcast requests between multiple, local area networks. Once the object has been located, its location is stored in a location cache so that the locate procedure need not be repeated unless the object is moved or that instance becomes inaccessible.

Through the use of these mechanisms, we now support remote file managers, although current bandwidth limitations and other problems with the Internet gateways have limited our experience in these areas. This approach has proven quite effective for providing remote access to line printers managed by various machines on different networks within the BBN complex. These mechanisms, with improved gateway support, will form the initial foundation for resource sharing between the BBN and RADC facilities, when the RADC facility becomes operational. As they exist now, these mechanisms do not restrict access to resources on a cluster beyond the normal Cronus object level access control mechanisms; for true inter-cluster operation, which crosses administrative boundaries, additional support for limiting foreign access to resources on a particular cluster will be necessary.

4. Resource Management

As a distributed system architecture, Cronus faces a number of resource management issues not present in non-distributed architectures. In this phase of development we have focused on the binding of a request from a client to a particular resource manager for those resources which are available redundantly. Redundancy comes in two forms: replicated objects and replicated managers. In both cases the selection of an object manager to provide the given service is an important resource management decision.

The general approach to resource management in Cronus is to individually control the management of the classes of objects which make up the system, rather than restricting all management of all resources to follow one particular, system enforced policy. This approach also allows Cronus resource management concepts to flow into the abstract model of resources managed by applications. In addition to system or service oriented resource management, application and system interface code can use the same mechanisms to implement policies that incorporate larger purviews of the resources, such as a policy which tries to optimize the use of collections of different objects types used in a particular context.

We have implemented mechanisms that allow resource selections to be made at two levels: by the client submitting the request and by the collection of managers responsible for a each type. The client may collect status information about the available managers using any available means (including the *report status* request) and then direct the invocation of an operation to a particular host or manager. The client specifies in the request that the operation must be performed at the specified host; no resource management decisions will be made by the manager itself in this case. If the operation cannot be performed by the manager at the selected host it will refuse the request and the client must choose a different manager to continue. Normally, requests do not identify a particular site for the invocation and the managers for the type of resource collectively make resource management decisions. The managers dynamically collect status information from their peers using the *report status* operation, and any other appropriate mechanism, and then forward the client request to the manager best suited to perform the operation. Thereafter, the manager to which the request was forwarded will process the request, as if it was the original recipient, and then reply directly to the client that originally issued the request.

To experiment with resource management and to test the mechanisms, we have modified the primal file manager to implement a resource management policy for creating new files. The mechanisms work as follows. An initial request to create a new file is routed to any available file manager based on the first response to a locate operation or a manager already in the kernel's object location cache. When a primal file manager at the selected host receives the file create request, it checks the local space usage and processor load. If either of these parameters exceeds operator selected thresholds, the file manager will not process the request itself. Instead, using status collected from the other managers it will choose the one it considers to be best suited to perform the operation. It then forwards the request to the selected manager for processing. The policy parameters that guide the selection can be set by the operator through the MCS operator interface or by invoking simple commands available elsewhere in the cluster.

These mechanisms also support other resource management policies used in the C^2 Internet experiment. For example, timer managers elect one manager as a reference clock, which is then responsible for periodically synchronizing the time of day clocks at the remaining timer managers. The parameters that control this selection can be set by the operator. The managers themselves keep statistics on how much their clocks drift from the reference clock so that when electing a new reference clock, the one with the least drift will be prefered. In another C^2 Internet experiment example, a collection of mission data managers share the work of storing target detections by examining each other's workload and rotating their responsibility for recording data for missions in a round-robin fashion.

5. Survivability Enhancements and Reconfiguration Support

A primary goal of the Cronus architecture is survivability in the face of system component failures. In the C^2 environment it is especially important to provide continuous availability of key applications despite system failures. There are two aspects of survivability which the Cronus architecture addresses: the availability of the system and its services over a relatively long period of time and the survivability of the applications which run on it. Application survivability is dependent not only on sustaining the application itself and the abstractions it presents to its users, but also on sustaining the resources on which it depends for its computational support. The object oriented approach taken in Cronus decomposes this problem into two parts, each of which must be made survivable: first, the objects and functions needed to sustain a computation, and second, the access path between the client and the objects or functions, must both endure partial system failures. We support object and function survivability through replication of the object instances and the managers which maintain and operate on these objects. We support survivabilty of the access path to these instances by detecting when an access path or object becomes unavailable, locating an alternative site with a copy of the instance, and automatically reconfiguring the access path to connect the client to the site with the available copy. In addition, in the event of an access point failure, the host independent nature of the application software makes it possible for the affected functions to be performed elsewhere in the cluster.

The support for survivability is delivered at three levels. At the low level, we provide mechanisms to support object replication and path reconfiguration: routines for recording the object instance descriptors in permanent storage now detect when an instance is changed and notify managers for the duplicates so that the change will be reflected there; mechanisms that support resource management provide the reconfiguration support needed to maintain an access path to an object through location independent UIDs, kernel based UID searches to locate instances of objects when their location is unknown and when their location changes, and caching these locations to improve performance. Second, we implement different replication strategies built upon these low level mechanisms to identify characteristics that distinguish different styles of object management for which different replication strategies are appropriate. And third, we allow the developer to select among various survivability properties offered in a high-level, language based interface.

Our first experiment was to introduce replication into the authentication manager. The authentication manager maintains descriptions of *principals*, representing particular people or other agents with whom access rights may be associated. It also maintains descriptions of *groups*, which are collections of principals and other groups which allow access rights to be given to many people at once, without knowing each member individually. For a cluster the size of our demonstration environment, two authentication managers which completely duplicate the entire principal and group databases is adequate. This provides adequate performance in our configuration, and a mechanism to independently control the replication of each object would have needlessly complicated the problem.

In this first experiment, changes to a principal or group object instance made by a manager at one site are broadcast to all other managers responsible for the same type. Each of these managers then apply the changes to their copy. Copies of a modified object are also updated whenever a manager is restarted--at restart time, the manager requests updates from its peers for all objects that have been modified while the manager was unavailable.

After an initial version of the replicated authentication manager was implemented it underwent a period of testing and revision. Then, use of the underlying replication mechanisms was added to the Cronus program support library and to the manager development tools. The authentication manager was then adapted to use these interfaces and tools, additional testing and evaluation were performed, and the mechanisms were released for use by application developers. A developer may currently choose whether objects of a particular type are replicated or not. He may also choose whether all objects of that type will be replicated, or whether only objects which an operator later identifies will be replicated. In the later case, replication can be independently enabled and disabled for each object, as the user desires. The mechanisms were subseq ently used in the C^2 Internet project to produce a replicated timer manager, that controls the progress of the experiment simulation. This particular manager uses the tool and library interfaces to replicate particular simulation clocks called *timers*, and uses the more primitive broadcast and forwarding facilites to elect a master clock which will then maintain the synchronization of the all the managers of timer objects.

A second experiment, to investigate a strategy that allows replication to be independently controlled for each object along with commands to support this control, now supports the survivability of the Cronus catalog manager. The Cronus catalog provides a system wide, user maintained, logical name space for use by people and application programs. In previous versions, global directory replication was limited to directories between the root and a boundary called the *dispersal cut*. While this was an effective strategy for ensuring the survival of directories which are shared and frequently needed, it was inconvenient to maintain because updates to this dispersal cut where manually introduced, coordinated and distributed by a system operator. Also, a duplicate of every directory between the root and the dispersal cut was kept by every catalog manager. In improving this original strategy, we wanted to mechanize the update process to eliminate the need for a specially skilled operator. We also wanted to allow the degree of replication to be varied for each directory, both to reduce the overhead associated with each directory update, and to allow a user to individually decide which directories need to be replicated and to tailor the number and placement of the directory copies.

The revised catalog manager supports these additional features. We have also developed user commands for controlling the replication and for reviewing the distribution of the copies and the availability of each of them. Unlike the authentication manager, the catalog survivability scheme is not currently available through the manager development tools. However, we anticipate that in the future, the manager developments tools will offer a choice of replication techniques, based on experience gained from those being used in the authentication manager and by the catalog manager.

6. Tool Integration and Distributed Application Development Support

Cronus is an extensible operating system. It has been designed to support a common structure for system and application components, so that mechanisms designed to aid distribution, resource allocation and reliability, and tools used to develop the operating system, may also be applied to developing distributed applications. We feel that the object oriented organization, used in developing the Cronus operating systems, may be extended into the application domain. That is, a new application may be organized and developed by identifying the types of objects involved and the operation protocols that objects of each type follow.

For an individual Cronus application developer, there are a number of functions that the implementer will perform repeatedly for each component he builds. For client software, these include routines for composing messages, sending them to the correct target, awaiting a reply, and parsing the reply to extract the requested information. For manager software, the needs include routines for recieving invocation requests, identifying the operation, parsing and extracting the parameters, dispatching to an appropriate routine to perform the operation, modifying the object instance descriptor and returning a reply to the client. The implementer must also develop software that supports various properties of the objects, such as access control restrictions and survivability. The software to handle these tasks is quite regular for most objects; this argues for automation of the process of producing support for these tasks.

The fact that distributed applications may span a wide geographic area and may require the cooperation of several independent individuals introduces additional need for formalizing interfaces between components. People responsible for different parts of an application must agree to interfaces before integrating their pieces; over time, the system may grow and the implementation change, but if we preserve the interfaces and their behavior, we can limit the effect of changes we make to the implementation. Also, the interface to an existing component can be used as a model for the interface to a new, but similar component.

As a first step toward satisfying these needs, we have introduced an interface specification language. For the individual, this allows the implementer to reuse code, increasing his personal productivity. For groups, this allows the members to specify interfaces among their components and have the system police individual compliance with the specifications. New components can often be fit into existing interfaces with appropriate support code. This language allows a designer or implementor to specify the properties of an object type, such as access control rights and survivability, and the parameterized operations that may be invoked on objects of the specified type. The language also allows *canonical types* to be specified. Canonical types provide a representation for data that is communicated between processes and stored by processes, regardless of the type of machine on which the processes are operating. Canonical types are also typically used to specify the variables that represent an instance of an object type.

Normally, an application developer will proceed by first identifying the functions to be performed by his application and the resources that will be employed in performing these functions. Then, object types will be specified for each kind of resource. Often, several resources will have very similar interfaces, as is the case with primal files, COS files, and the line printer interface. In such cases, a *parent type*, which specifies these common interface characteristics can be defined and the other types will be made *subtypes* of this parent type. The subtypes inherit the same definition for the common operations, as defined by the parent, without the need to be repeatedly specified.

6.1. Development of New Types

Designing a new type involves characterizing the role of objects of the new type in the application, specifying operations that affect the status or behavior of objects of the new type, and specifying the descriptor for instances of the object type. Support for easily adding new types has already involved work in several areas, including specification, code generation, distribution, and debugging support. These areas have been explored by first experimenting with facilities for use with system objects, generalizing the interfaces, verifying the results with system components, applying the results to applications needs and evaluating their effectiveness.

One problem with supporting a new type within the system is producing the code which implement object managers for the type. Much of this code has a regular structure. When invoking an operation, the client creates a message buffer, identifies the operation to invoke, specifies the parameters, uses the invoke primitive to transmit the message to a manager for the target object, awaits a reply, parses the reply and processes the results. At the manager end, the request must be parsed, access control checks made, the proper operation processing routine called, the results placed in a message, and this reply message returned to the client. Multi-tasking support allows multiple requests to be in-process concurrently. Object descriptions must be kept in stable storage to record the current state of each object for which the manager is responsible.

Our initial approach to the problem of constructing new type managers includes code generation and library support. The implementer produces a specification of the operations that are appropriate for objects of the new type. The new type may also be made a subtype of another type, causing the operations of the parent type to be inherited by the new type. The operation specifications include access control checks and a description of each parameter; canonical types may be introduced to specify new types used to communicate information between client and manager. From this specification, code for both the client and the manager is produced: client code provides subroutines that package an appropriate request, invoke it on a specified object and return the results to the caller of the subroutine; manager code handles message parsing, access control checks, dispatching to operation processing routines and returning the reply to the client. The tools also provide library support for facilities used regularly by managers: multi-tasking, access control list modification, maintaining data associated with object instances, and support for generic operations such as *locate*.

The distribution and sharing of these specifications introduce unique problems in a distributed environment since clients may be on different hosts than the managers, and managers may be developed on different hosts than their parent types. To maintain the information about Cronus object types in a globally accessible way, we have implemented a *type definition manager*. After a developer has specified the operations and interfaces for a new object type, he invokes an operation to transmit this information to the type manager. Thereafter, the information can be retrieved from the type manager by any client in the cluster. The development tools which generate code for clients and managers retrieve this information and produce machine independent code for compilation and link editing on appropriate target machines. This information can also be read dynamically by programs before invoking an operation, as is done by the user interface program *ui*: this program retrieves the specification for a selected command and then interactively prompts the user for parameters.

To speed development, support for many generic operations is supplied by the tools. Type definitions are supplied for generic object operations appropriate to all objects, such as *locate* and *remove*, and for generic operations on replicated objects, such as *replicate*, *dereplicate*, and show changes. A developer may include these in a new type by naming the *object* or *replicated object* type as the parent for his new type. Library support provides routines for performing these operations.

To date, these development support tools have been used in producing several system managers, including the authentication manager, COS directory and file managers and line printer manager. It has also been used in producing all managers used in the C^2 Internet experiment.

6.2. Software Distribution Manager

Software distribution in a distributed development environment, though seemingly simple, often becomes an extremely complicated, time consuming, and error-prone task. Our primary goal was to provide a simple abstract model to the developer and to limit the amount of information the developer has to understand and manipulate. The volume of information a developer might have to consider is potentially large, especially if all appropriate (*file, site*) pairs must be identified. Our approach is to group files with identical distribution requirements into *packages*, and to identify to which sites each package should be distributed. This representation is more natural to the developer of large applications, since such a user will normally think in terms of collections of files composing a distributed application or subsystem, and this representation provides a much more concise and intelligible description of the distribution requirements than listing the individual (*file, site*) pairs.

The distribution process is controlled by a logically-centralized manager process, rather than independently from a variety of client programs. This has the benefit of limiting knowledge of the implementation of packages to one program, making it easier to enhance, and of minimizing the interface requirements at the user access point, since the user need only be able to invoke a single Cronus operation. Since the package data is only slowly changing, it is easily replicated to avoid any single site outages preventing distribution. The package manager transmits each file to the target host by using the Constituent Operating System (COS) Interface Manager on the host. Thus, adding a new site bearing host only requires the development of COS Interface Manager on the new host, a service normally provided anyway.

For the initial version, we decided not to automatically trigger updates. This would have required additional mechanisms to identify when a set of changes were actually suitable for distribution. Instead, developers explicitly initiate updates when they are confident that the files represented by a package are consistent. In the future, a daemon process might regularly look for changes at a designated primary site and automatically initiate updates. For now, we are primarily interested in providing the essential mechanisms and conceptual framework for managing the contents of the packages.

Initiating component distribution and maintenance of the file lists and sites are independently access controlled to reflect the differing roles of software developers. Developers modify the implementations, affecting the contents of particular packages. System administrators determine the ultimate placement of support for the services and application components.

The design and implementation of the software distribution componeat makes extensive use of existing Cronus facilities. The Software Distribution Manager was constructed using the manager development tools, and invokes operations on other managers using the automatically generated program support library subroutines. The manager is not dependent on the contents of the files in a package: they may be source files, language processor header files, command scripts, or, when distributed between hosts of the same type, binary executable and library files.

6.3. Development Tools: Editors, Compilers and Utilities

Cronus is both a base operating system for supporting distributed applications and an environment for developing these applications. One important aspect of supporting software development in a distributed environment is a distributed file system. A distributed file system is useful only to the extent that there are tocls which can utilize the distributed file system. An initial step toward making Cronus more useful for software development is to provide a set of development tools which utilize Cronus functionality. Such tools include editors, compilers and linkers.

At the outset, we have chosen to adapt existing tools to the Cronus environment whenever possible, rather than developing tools specifically tailored for the Cronus environment in order to gain at least some immediate functionality. To reduce the effort required to adapt existing tools, we have modified the file system subroutine libraries for the VMS, C70 Unix and Vax Unix systems. These *trap* libraries invoke Cronus operations whenever a file name specifies a Cronus file. Otherwise, they behave as they did before modification, performing the operations on VMS or UNIX files. The VAX-UNIX trap library was extended during the first part of this contract and has been used to produce several UNIX based file utilities as mentioned in an earlier section.

To simplify use of electronic mail for communication among members of particular groups, particularly for suggesting changes and notifying people when changes occur, we have implemented a mail program that allows messages to be addressed to Cronus principals and groups. This has the added benefit of supporting distributed maintainance of mailing lists.

6.4. Distributed Access to Constituent Operating System File Systems

Through Cronus, it is also desirable to gain remote, distributed access to directories and files maintained by a Constituent Operation System (COS). This allows remote access to mailboxes, bulletin boards, on-line manuals and other data whose contents are customarilly maintained by COS utilities, but for which access should be available throughout the cluster. We have implemented a manager, called the *COS Interface Manager*, which provides access to directories and files stored on the COS. Registering a COS file or directory with this manager returns a Cronus Unique Identifier (UID) that can be later used to manipulate it remotely as a Cronus object and through the Cronus catalog

The operations of COS directories and files emulate those of the Cronus catalog and Cronus files so that, in most cases, users need not be aware of whether a particular catalog entry refers to a Cronus primal file or a COS file. Thus, the Cronus utilities, such as *display* and *list* directory, work with COS files and directories as they do for compatible Cronus objects. The COS Interface manager is a step in the gradual evolution between completely independent host systems and a completely integrated distributed system.

7. RADC Cluster Support

An important part of demonstrating the applicability of Cronus in the C^2 environment, evaluating its capabilities, and successfully transferring DOS technology is the installation and operation of a Cronus DOS cluster at RADC. Doing this will provide valuable experience in transporting Cronus to another environment and seeing how well it can be operated and used by a different user community. The Cronus cluster at RADC will be gatewayed to the DARPA Internet so that it can be accessed remotely from the cluster at BBN. This will allow both remote operation and monitoring of the RADC cluster and experimentation with inter-cluster operations in the Cronus DOS.

We have been assisting RADC with the selection of the hardware configuration for the Cronus cluster. In order to facilitate installation and operation of the RADC cluster, our major guideline in the selection has been compatibility with the BBN cluster, at least in terms of the types of machines and operating systems supported and the underlying local network. We produced a cluster installation report that details how to install Cronus once the cluster hardware has been installed. RADC is in the process of acquiring and installing the necessary components.

8. Cluster Maintenance

The major maintenance effort in this area was transporting the existing GCE system to a more modern, commercially supported hardware base. The processor, peripherals and other components of the original GCE system where purchased from commercial sources and integrated by project staff members several years ago. While we were able to maintain and repair the system, using commercial field support when available, and performing the repairs ourselves otherwise, the peripheral devices were failing more frequently. Compared with commercially available microcomputer systems that have been produced in the past few years, our equipment was difficult and costly to maintain, To correct this problem, we evaluated several microcomputer systems, looking for a suitable replacement. We selected a MassComp 68000 based system and transported Cronus to it. This machine, and possibly similar products from other vendors, will form the hardware base for future GCE development.

To reduce the overhead involved in routing and forwarding large volumes of data between Cronus processes, we have fixed and enhanced the Cronus large message transfer mechanisms. Formerly, it was necessary to send large amounts of data using a series of small messages; in addition to being processed by the sending and receiving process, each small message would pass through the Cronus kernel at both the sending and receiving site. Now, large messages are transmitted via a TCP connection between the sending and recieving process. The Cronus kernels are still responsible for sending the initial portion of the message, which contains the information necessary to establish the connection, but thereafter, all data is passed directly between the processes. The choice of message type is made by the program support library routines and the connection is established with the cooperation of the kernel. The sending process need not be aware of which mechanism will be used, nor need it advise the routines which mechanism to use. As long as the buffer supplied by the recieving side is large enough to contain the data, it need not be aware of which mechanism is being used; if the receiving buffer is too small, the message will be delivered in a series of consecutive pieces. A similar mechanism is provided on the sending end to allow it to send a message as a series of smaller pieces.

As the size of our cluster has grown, keeping backups of storage system has become an increasingly time consuming task. As part of our maintenance effort, we have developed an initial version of a primitive backup and restore facility which allows Cronus objects to be copying to magnetic tape and later restored. Using this facility allows us to dump sources and other valuable files from any Cronus host to a single central location.

Performance and reliability of the *Compion* TCP/IP software, used under VAX-VMS, proved to be unsatisfactory. We have replaced it with software provided by *Wollongong*, and have been successfully using it for several months.

9. Papers and Technical Articles

Several papers describing different areas of the Cronus development have been written; a number have already been accepted for presentation at conferences. In this section we summarize the various papers. Copies of the papers are included either in the appendix to this report or in the appendix to *Interim Technical Report No. 5**.

9.1. ICDCS Papers

Two papers, included in Interim Technical Report No. 5 as Appendix A and Appendix B, have been accepted for presentation at the Sixth International Conference on Distributed Computing Systems, to be held in May 1986. The first of these, *The Architecture of the Cronus Distributed Operating System*, describes the overall architecture of Cronus and details the design of key components of the system. The second paper, *Programming Support in the Cronus Distributed Operating System*, presents our approach to the problem of distributed application development, describes the features of Cronus that support this development, and illustrates how Cronus facilitates development using a Cronus object manager as an example.

*Cronus, A Distributed Operating System: Interim Technical Report No. 5, *R. Schantz, et. al.* Technical Report No. 5991. BBN Laboratories Incorporated, June 1988, RADC-TR-88-132, Vol III.

9.2. Use of Canonical Types

The Cronus Distributed Operating System has as one of its major goals the exchange of information among heterogeneous hosts in an Internet environment. To allow this, a common data representation must be adopted for network traffic. This paper describes that representation and its application. Cronus canonical data types are extensible; new types may be constructed via a high-level definition language. This ability, along with program development tools that automatically generate code for data conversions, virtually eliminates representation considerations in the development of Cronus distributed applications.

This paper has been submitted for presentation at the ACM SIGCOMM '86 Symposium on Communication Architectures and Protocols. A copy of it is included in this report as Appendix A.

9.3. Constituent Operating System Integration Guidelines

Integrating new hosts into Cronus is one of the long term objectives for the system. This article discusses the issues surrounding the integration of Cronus with a constituent operating system (COS). In order to support different degrees of COS integration, Cronus has facilities have been designed in layers. It is intended that a minimal integration may be achieved by implementing only the lowest layers, and that greater degrees of integration can be added incrementally. This approach has been experimentally tested by integrating a number of COS systems and host architectures into the BBN Cronus cluster. The experience obtained from implementing Cronus on this variety of hosts and operating systems in the Cronus test configuration forms the basis of the information in this article.

A copy of this article is included in this report as Appendix B.

9.4. Ethernet Experience

The Cronus project has owned and operated an Ethernet for several years. The network currently provides services for over twenty hosts whose resources are either used directly or indirectly by Cronus project members, and provides access to the DARPA Internet and other networks through gateways. It has been our experience that all of these disparate hosts can be made to coexist and intercommunicate on the same Ethernet. However, this intercommunication capability has been achieved only after considerable effort. We have written a note which tries to capture some of the experience that has been accumulated in managing the Cronus Ethernet, and to indicate the types of problems which can be expected in the management of similar local area networks.

A copy of this article is included in this report as Appendix C.

9.5. Broadcast Repeater RFC

The paper was included in Interim Technical Report #5 as Appendix C; it has now also been distributed as an Arpanet RFC 947. It describes the extension of a network's broadcast domain to include more than one physical network through the use of a broadcast packet repeater.

A. Use of Canonical Types

Canonical Data Representation in the Cronus Distributed Operating System

Michael A. Dean
Richard M. Sands
Richard E. Schantz

BBN Laboratories Incorporated
10 Moulton Street
Cambridge, Massachusetts 02238

Abstract

The Cronus Distributed Operating System has as one of its major goals the exchange of information among heterogeneous hosts in an internet environment. To allow this, a common data representation must be adopted for network traffic. This paper describes that representation and its application. Cronus canonical data types are extensible; new types may be constructed via a high-level definition language. This ability, along with program development tools that automatically generate code for data conversions, virtually eliminates representation considerations in the development of Cronus distributed applications.

A.1. Introduction

Cronus is an object-oriented distributed operating system under development since 1981 at BBN Laboratories. It differs from other DOS projects in its use of a heterogeneous computing base, and emphasis on interoperability with existing constituent operating system resources and facilities. Our current configuration includes BBN C/70s, DEC VAXs running both Unix and VMS, and Sun and Masscomp workstations – integration of symbolic and parallel processing elements are planned for the near future. We believe that the distributed application developer shouldn't have to give up familiar computing environments, programming languages, tools, and utilities. To such a user, the Cronus object model and its protocols present a uniform interface to diverse computing resources.

This paper describes a key feature of this uniform interface, the Cronus canonical data representation scheme. This system component defines and implements a set of common data representations used to exchange data among the elements of a distributed application in a heterogeneous environment. A more complete discussion of the overall architecture of Cronus can be found in [Schantz], an overview of the application development environment can be found in [Gurwitz], and a prototype application is described in [Berets].

Cronus development has been supported by the Rome Air Development Center, under contracts F30602-81-C-0132 and F30602-84-C-0171.

A.2. Background

One of the fundamental problems inherent in integrating a varied collection of computer architectures and local operating systems is the differing representations that these systems may use in storing and manipulating data. On the systems which Cronus is currently implemented, integers may be signed or unsigned, and depending on the word size of the system, are 16, 20, 32, or 40 bits long. Some of these systems store the low-order bytes of an integer in lower memory addresses than the high-order bits, and others store the high-order bytes of an integer in lower memory addresses. All of these systems use two's complement integer representations, but there are systems which use BCD, one's complement, and signed magnitude representations as well. This situation is not limited to integer data types; nearly all common basic data types have several popular representations.

Each system and the languages used to program on those systems are able to efficiently process information only in the representations which they directly support, usually with hardware. Data interchange between systems with differing representations will therefore require translation from one data representation to another.

One obvious technique for solving this problem would be to communicate the data from source machine to destination machine in the source machine's data representation, with the destination machine responsible for converting the data to its own representation, based on the origin of the data. Similarly, the source machine could convert the data to the destination machine's format before transmission, based on the destination's architecture and operating system. Either of these techniques has the advantage that for transfers between similar machines, no data conversions are needed. There are a number of disadvantages to these methods however that far outweigh this one advantage.

One problem is that there must be two data conversion routines for each pair of representations of some data type, available to all programs that will communicate data of that type. This results in a very large number of routines that must be maintained, debugged, and linked as part of each program's executable image. Another problem is that adding a new machine type would necessitate relinking all programs on all machines, since the new data representations would require additional conversion routines in all programs.

Another solution to this problem presents itself upon examination of the defects of the previously mentioned techniques. A canonical data representation for each data type could be invented, and all data interchange would involve conversion of data from the originating machine's data formats to canonical formats, transmission of the data in canonical form, and conversion from canonical representations to the destination machine's data formats at the destination.

With this technique, each system architecture would require only one pair of conversion routines per basic data type for converting between its own data representations and canonical representation. This results in a much more manageable number of conversion routines. When a new machine type is added, no relinking of programs would be necessary, since data would be received in canonical format no matter what the architecture of the originating system.

Cronus uses the technique of canonical data representation to solve the problem of data interchange in a heterogeneous computing environment. This method is far simpler to implement, and more flexible than any technique involving communication of data in machine dependent form. Programs process data in formats directly supported by the systems on which they are implemented. When data is transferred to another network component, it is encoded into a canonical form using appropriate conversion routines. The reverse process takes place on the receiving end.

In Cronus, we take the data exchange problem a step further by incorporating it into the basic computational model developed for the system. Cronus models operating system and application resources as abstract objects, and access to resources as invoking operations on those objects. Operation invocation is implemented using messages exchanged between the invoking client program and an object manager program which performs the operation on the data object. It is often the case that the client and the object are on different computer systems, which exhibit significant differences in architecture, operating system, and data representations. When the communicating systems are different, care must be taken to insure that the data and control messages exchanged are understandable to programs on both sides.

Programming support in Cronus is based on the assumption that heterogeneity will be common in the applications which Cronus has been designed to support, and that it is desirable to provide application developers with an environment in which such heterogeneity is handled without directly involving the programmer. Although assuming that communicating components are different may appear to be a worst-case assumption, we believe that it is easier to plan for heterogeneity at the outset rather than adding support for it to an already existing system. If necessary, Cronus can be optimized later for the homogeneous case.

A.3. Canonical Types

Fundamentally, a Cronus canonical data type includes a canonical representation expressed as a sequence of 8-bit octets, a set of internal representations, one for each target programming language, and a set of subroutines for converting between internal and external representations and determining the amount of memory which must be allocated to contain them.

Consider, for example, the canonical type U16I, used to represent unsigned integers in the range 0 to 2**16-1. Its canonical representation consists of two octets containing the most-significant and least-significant 8-bits of the number, respectively. It is represented in C language [Kernighan] programs as "unsigned int". Note that the internal representation must be able to represent all possible values of the canonical representation, but the corresponding canonical representation may not be able to represent the entire range of the internal representation. The conversion routines report an 'argument out of range' error if an attempt is made to convert an internal value that is out of the range of the corresponding canonical type. For flexibility, there are signed and unsigned 16 and 32 bit integer canonical types available, called U16I, S16I, U32I, and S32I.

U16I is a "fixed length" canonical type, which means that both the canonical and internal representations require a constant amount of storage, whatever the value represented. ASC (ASCII character string, represented in C by "char *") is an example of a "variable length" canonical type, in which both the canonical and internal representations require variable amounts of storage, depending on the value represented. If the amount of storage needed to represent a data type can vary, then the corresponding canonical type will be variable length as well. Variable length canonical types always consist of a length field in octets, encoded as a U32I, followed by the variable amount of data making up the canonical representation. In addition, all canonical types consist of an integral number of octets. Other than these two restrictions, any encoding of an internal representation's range into octets is acceptable.

There is a set of standard system-supplied canonical types that represent both common programming language data types such as integers and booleans, and Cronus system types such as Unique Numbers (UNOs). In addition, there is an ARRAY canonical type that can represent a one-dimensional array of any other canonical type. A partial list of the current standard canonical types, their lengths, and the corresponding internal representation in C is given in Figure 1.

Canonical	C Language	Length	Description
Type	Representation		(octets)
EBOOL	int	1	Boolean true or false. The internal values are limited to false = 0, and true = 1.
U16I	unsigned int	2	Unsigned 16-bit integer
S16I	int	2	Signed 16-bit integer
U32I	long	4	Unsigned 32-bit integer (C does not allow a declaration of 'unsigned long')
S32I	long	4	Signed 32-bit integer
ASC	char *	variable	ASCII character string
OVEC	struct OctetBuffer	variable	Self-defining octet vector
EDATE	struct DATE	8	Date/Time
EUNO	struct UNO	10	Cronus unique number
CTYP	typedef int TYPE	2	Cronus object type identifier
EUID	struct UID	12	Cronus unique identifier
ARRAY	<cantype> *	variable	One-dimensional array of some other canonical type
MS	struct OctetBuffer	variable	Message Structure

Standard Canonical Types
Figure 1

Canonical representations generally are quite straightforward, with simple mappings between internal and external values. Usually, structured canonical types follow organizations similar to their internal data type counterparts. For example, the canonical representation of an integer called 'value' converted to a U16I is:

value / 256	value mod 256
1 octet	1 octet

If value is 1859, its canonical representation (2 octets, in hexadecimal) would be 07 43.

As another example, the canonical representation of an ASCII character string is:

length	char[1]	...	char[length]
4 octets	length octets		

where length is a U32I whose value is the number of characters in the string, and successive octets after the length contain the 7-bit ASCII characters of the string, with the high-order bit 0. The canonical representation of the string "Cronus" would be 00 00 00 06 43 72 6f 6e 75 73.

A more complex canonical type is the ARRAY aggregate type, for which there are two representations: one for fixed length array elements, and one for variable length array elements. Arrays of any canonical type can be represented.

The canonical form of fixed length item arrays consists of a type field followed by the canonical forms of the items in the array, back-to-back in a contiguous string of octets. Lower indexed items, starting with item[1], come before higher indexed items in the octet vector:

length	cantype	item[1]	...	item[n]
4 octets	2	length(item)*n		

The canonical form of variable length item arrays consists of a type, the number of items in the array, a block of descriptors giving the integer offset of each item in the following octet string, and a string of octets containing the variable length items back-to-back. The lengths of the variable-length items are encoded in the first four octets of the items themselves. A variable length item array could be encoded as:

length	cantype	n	offset(item[1])	...
4 octets	2	2	4 * n	

...	offset(item[n])	item[1]	item[2]	...
...		length(item[1]) + ...		

...	item[n-1]	item[n]
... + length(item[n])		

Each canonical type <type> is defined by a pair of functions, one for converting the internal representation to canonical form, called To<type>, and one for converting the canonical representation into internal form, called From<type>. For example, the two conversion functions for U16I are called ToU16I and FromU16I. Variable length canonical types have two additional functions for determining the canonical length of an internal value, and vice-versa, used for memory allocation and message sizing. Because these functions are usually called from other routines that do not know in advance which types they will be required to convert, these functions are required to follow a strict interface convention. This convention also makes it possible to dynamically add new canonical types to the set of known types without changing higher levels of software.

Because Cronus is designed for a heterogeneous computing environment, the conversion functions for the standard canonical types have been written to be portable to any machine architecture. They do not depend upon the specific word length or byte ordering of any specific machine, and work without modification on 10 bits per byte BBN C/70 minicomputers, on MC68000-based workstations which require integers to be word-aligned with the most significant byte first and on DEC VAX minicomputers with the opposite byte ordering.

New composite canonical types can easily be built out of existing ones. For example the Cronus canonical type EUID, a unique identifier which is used for naming objects, actually consists of a EUNO (unique number) and an object type.

EUNO	CTYP
10 octets	2

In addition to arrays, Cronus also supports lists of items, via the Message Structure constructor. This is an octet vector containing an unordered sequence of key/cantype/value triples.

key	cantype	value
2 octets	2	depends on cantype

The key serves as a name for the following type/value pair. Keys are represented as U16Is. Individual fields within a Message Structure can be referenced by passing the desired key to a collection of library routines for manipulating Message Structures. Values are always presented to application programs in the internal format dictated by the canonical type.

Message Structures, as the name suggests, are used to build messages in Cronus. Each message contains a set of standard fields specifying the target object, operation, and a transaction identifier, along with any fields defined by the application. An example of the Message Structures used to invoke a sample Cronus operation and its corresponding reply is given in Figure 2. This operation creates and initializes a new Group object, returning its unique identifier. The figure notes which of the keys are standard (found in all Cronus invocations and replies), and which are unique to this application.

The ARRAY and Message Structure canonical types are examples of a class of canonical types called constructors, which aggregate elements of other canonical types. These are fully recursive and extensible, allowing arbitrarily complex structures to be represented. Additional constructors, such

Key (std/applic)	Cantype	Value
MsgType (std)	U16I	Invocation (0)
Object (std)	EUID	{group}
OperationName (std)	ASC	"Create"
TransactionID (std)	EUNO	{bbn-vax:153:219}
InitialMembers (applic)	array of EUID	Dean, Sands
MsgType (std)	U16I	Reply (1)
TransactionID (std)	EUNO	{bbn-vax:153:219}
NewObject (applic)	EUID	{bbn-sun1:47:91:group}

Sample Invocation and Reply Messages
Figure 2

as Binary Tree, could easily be defined for applications requiring them.

A.4. Automated Definition of New Canonical Types

With Cronus, we've sought to automate as much as possible of the distributed application development process. The cornerstone of this approach is a high-level non-procedural specification language for defining the operation invocation protocols for new Cronus object types. At the definition level, Cronus types are similar to Smalltalk classes [Goldberg]: a set of abstract operations are specified for each object type, children inherit operations from their parent in the type hierarchy, and each operation contains some number of parameters (fields). Figure 3 contains a partial specification for a Cronus object type TextFile, having traditional file semantics. A specification has three parts: the definition of the type, including parameters governing operation inheritance and access control, a specification of the operations to be supported and their parameters, and the definition of any new canonical types. Note that unlike Smalltalk, parameters in Cronus operations are strongly typed, using canonical data types. Often a developer will want to use more than just the system supplied canonical types for these parameters. We therefore provide a mechanism for specifying new canonical types in terms of existing canonical types. From these specifications, implementation representations and conversion and sizing functions are automatically generated.

Because Cronus is targeted for a heterogeneous computing environment, it is important for our interface specifications to be independent of any particular programming language. For specifying new canonical types, we selected the smallest set of constructs which would provide sufficient expressive power and yet be typesafe and directly representable in most modern programming languages. We chose 1) enumerations, and 2) records (structures) made up of previously defined canonical types and unbounded arrays of previously defined canonical types.

As an example, consider the definitions of the FILESTATUS canonical type in Figure 3, which is returned by the Status operation. This contains the length of the file, the last time it was modified, and a list of the processes which have Opened the file but not Closed it. The latter includes the type of access (OpenMode) demonstrating the use of an enumerated canonical type.

Type specifications serve as the basis for automatically generating much of the distributed aspects of the application. In addition to the functions implementing the defined canonical types, we generate manager code to parse, decode, and validate messages, retrieve the permanent storage associated with the object, perform access control checks, and dispatch to a developer-supplied operation processing routine. For the client, we generate subroutine interfaces encapsulating the invocation message construction, data conversion, operation invocation, and reply message parsing functions. On both sides, the developer is provided with a simple typesafe interface using exclusively internal data representations.

A.5. Relation to Other Work

Discussions of problems and solutions with communication in a heterogeneous computing environment are by no means new. BBN's experience in this area dates back to the National Software Works project beginning in 1975 [Forsdick, White].

DeSchon [DeSchon] provides a brief comparison of several data representation systems currently in use in the DARPA Internet. Of those, Cronus canonical types are most similar in their structure and application to the Xerox Courier [Xerox] and Sun External Data Representation [Sun] protocols. In fact, both the Xerox work and the Cronus work are derived from experience with the NSW system. The Xerox and Sun approaches however, were designed with specific languages and computing environments in mind; with Cronus we have tried to take a more general view. Cronus is already used in the Unix computing environment for which Sun XDR was designed, and we believe that environments like Xerox Mesa/Pilot could also easily be integrated.

The Cronus mechanisms could also be compared to Accent IPC [Rashid], where conversions are performed within the communications kernel. This approach affords some optimization by eliminating conversions if the target host is known to be of the same type as the sender, at the cost of increased size and complexity of the kernel. The Cronus approach of making these non-kernel functions, affords flexibility and simplicity. This is especially important in achieving another project goal of keeping the size of the Cronus kernel to a minimum.

We believe that the most significant contribution of Cronus to the data representation area is in its demonstration that these problems can be solved sufficiently to make them a non-issue for most application developers. Almost all data conversions take place below the level of application code in Cronus, and very few of our developers are intimately familiar with the material presented in this paper. We believe such hiding is good, both for simplifying the development process and by making it more convenient to accommodate future changes and optimizations to the data representation and encoding facility.

```
type TextFile
        subtype of Object
        rights are read, write, append
        generic rights are create;

generic operation Create()
        returns(NewObject: EUID)
        requires create;

operation Open(For: OPENMODE);

operation Close();

operation Read(MaxLength: U16I)
        returns(Data: ASC)
        requires read;


...

operation Status()
        returns(Status: FILESTATUS);

cantype OPENMODE
        representation is OpenMode: { read, write, readwrite };

cantype OPENSTATUS
        representation is OpenStatus:
        record
        process: EUID;
        mode: OPENMODE;
        end OPENSTATUS;

cantype FILESTATUS
        representation is FileStatus:
        record
        length: U32I;
        lastmodified: EDATE;
        currentusers: array of OPENSTATUS;
        end FILESTATUS;

end type TextFile;
```

Sample Type Specification
Figure 3

A.6. Experience

Cronus has been running at BBN for over 2 years. Our current network includes 18 hosts of 6 different types. As of this writing, 43 different object types have been implemented, providing both system and application services. These types define 265 operations and 118 new canonical data types.

The Cronus canonical type mechanism has evolved over time. Originally, we assumed that most of our needs would be met via the primitive types and message structures (with lists of items in message structures playing the role that records do now). Messages were constructed and parsed by storing and retrieving key/type/value triples explicitly in message structures. We soon realized that, given high-level definitions of the operations and their parameters, interface routines providing a remote procedure call-like interface to operations could be machine-generated. The production of new canonical data types could also be automated in a similar fashion. The resulting record structures have significant advantages over message structures in terms of space efficiency and the ability to perform static type checking.

Message structures are still the building blocks for messages, but their use is almost completely encapsulated within machine-generated software.

We have found applications for canonical data types beyond their use in messages. In particular, we also use them to store the application-dependent "instance variables" associated with Cronus objects. This type of usage originally began as an expedient method of linearizing complex internal data structures for storage on disk. This initial idea has been extended by application-independent software for manipulating and transporting these canonical representations to support backup/restore, debugging, and transparent object migration and replication.

Cronus has so far concentrated on applications written in the C programming language [Kernighan]. Language heterogeneity has from the outset been one of our objectives, and some early components were written in Pascal to demonstrate language interoperability. We anticipate integrating support for other languages, in the new future, including Lisp and Ada.

Overall, we believe our support for data representation in a complex distributed environment has been quite successful. Heterogeneity in our computing base is no longer a significant issue.

A.7. Conclusions

The Cronus Distributed Operating System relies heavily on an extensible data representation scheme to facilitate communication in a heterogeneous multi-language distributed environment. New canonical data types are specified as part of the high-level interface definitions for new Cronus services. From these definitions we can automatically generate code implementing most of the distributed aspects of these programs.

All data is presented to the application in internal representations appropriate for the target programming language. By eliminating the need to write code for handling heterogeneity and application distribution, which tends to be repetitious and error prone, developers are freed to concentrate on their applications, using familiar languages, tools, and environments.

Data representation issues are no longer a major concern for developers of Cronus programs.

Acknowledgments

We would like to acknowledge the help of all of the people associated with the Cronus development effort for their direct influence on the design of the data representation facility and for their indirect validation of the approach through usage. The contributions of Dr. William MacGregor, who initiated the design, and Dr. Ben Woznick and Mr. Harry Forsdick, who suggested a number of key improvements, are especially noteworthy. Mr. James E. White deserves special mention for providing the influence of the earliest treatment of canonical data representation issues of which we are aware.

References

[Berets] J. Berets, R. Mucci, and R. Schantz, "Cronus: A Testbed for Developing Distributed Systems". *Proc. IEEE Military Communications Conference*, IEEE Communications Society, Oct. 1985, pp. 409-417.

[DeSchon] A. DeSchon, "A Survey of Data Representation Standards", RFC 971, DDN Network Information Center, Jan. 1986.

[Forsdick] H. Forsdick, R. Schantz, and R. Thomas, "Operating Systems for Computer Networks", IEEE Computer, Jan. 1978.

[Goldberg] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation.* Reading, MA: Addison-Wesley, 1983.

[Gurwitz] R. Gurwitz, M. Dean, and R. Schantz, "Programming Support in the Cronus Distributed Operating System". *Proc. 6th International Conference on Distributed Computing Systems*, IEEE Computer Society, May 1986, to appear.

[Kernighan] B. Kernighan and D. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

[Rashid] R. Rashid and G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel". *Proc. 8th Symposium on Operating System Principles*, ACM, Dec. 1981, pp. 64-75.

[Schantz] R. Schantz, R. Thomas, and G. Bono, "The Architecture of the Cronus Distributed Operating System". *Proc. 6th International Conference on Distributed Computing Systems*, IEEE Computer Society, May 1986, to appear.

[Sun] "Extended Data Representation Reference Manual", Sun Microsystems, Sept. 1984.

[White] J. White, "A High Level Framework for Network-Based Resource Sharing". *Proc. AFIPS Conference*, Volume 45, 1976.

[Xerox] "Courier: The Remote Procedure Call Protocol", Report XSIS-038112, Xerox Corporation, Dec. 1981.

B. Constituent Operating System Integration

Cronus COS Integration

Girome Bono

BBN Laboratories Incorporated
10 Moulton Street
Cambridge, Massachusetts 02238

B.1. Introduction

This note is a discussion of the issues surrounding the integration of constituent operating system (COS) into Cronus. Cronus is a distributed operating system that provides a uniform programming and user environment across a heterogeneous set of computers. It is the focus of an ongoing research project designed to investigate the feasibility of building an environment for distributed applications. One of Cronus' goals is to make the resources associated with each COS available in a consistant and controlled way to all of the other machines on the network.

Cronus has already been integrated onto a number of COS and host architectures. The test configuration at BBN includes several different UNIX implementations (4.2BSD, V7, SYSV), VMS, and CMOS running on VAXes, C70s and 68000s all connected by a Xerox Ethernet. The experience obtained from these implementations has been used as the basis for this paper.

There are many ways to integrate a new COS into a Cronus configuration. Minimally, the goal is to support the Cronus interprocess communication primitives on the COS. This allows local programs to access Cronus resources by invoking operations on them. A complete integration, however, may involve supporting standard Cronus user interface and program development tools on the system, and may also include implementing Cronus services on the system to allow remote access to its resources.

In order to support different levels of COS integration Cronus has been designed in layers. It is intended that a minimal integration may be achieved by implementing only the lowest layers, and that adding each new additional layer adds new capabilities and functionality to the COS. These layers are not strictly ordered, although there are some dependencies noted below.

<table>
<tr><td colspan="2">COS/Application</td><td rowspan="2">User
Interface
Programs</td></tr>
<tr><td>Resource
Managers</td><td>Manager Support
Library</td></tr>
<tr><td colspan="3">Cronus Kernel</td></tr>
<tr><td colspan="3">TCP/IP Protocols</td></tr>
<tr><td colspan="3">VLN Interface</td></tr>
</table>

The bottom three layers constitute a minimal Cronus integration. The first steps in integrating a host into Cronus are to connect to the particular LAN for the cluster and to provide flexible IP/TCP implementations to support the Cronus implementation. The virtual local network (VLN) provides a minimal set of datagram communication facilities, transmitting messages addressed to particular recipients and messages broadcast to all available recipients. A VLN may be a simple local area network, such as an Ethernet, or it may be several networks connected via standard Arpanet gateways. We currently use these VLN facilities through DoD's standard communication protocols (TCP/IP), which provides addition facilities such as reliable transport and connection based communication, although Cronus requires far less and can be adapted to use other transport protocols. The Cronus Kernel provides object addressed ICP, and performs simple process management functions.

The higher layers consist of libraries and tools to provide a programming and user environment on a Cronus system. These may be customized to the particular needs on the target COS. In particular the user interface to Cronus tends to be geared towards the kind of environment with which COS users are familiar.

B.2. Integration Procedure

The layers in Cronus provide a framework for a particular integration effort. The procedure for dealing with a new COS is first to determine what level of integration is desired and then to implement the necessary Cronus layers to reach that level. The following sections functionally describe each layer starting at the bottom and going up.

B.2.1. VLN Interface

The VLN interface consists of both the hardware and software necessary to access a host's physical network. Cronus requirements are defined as VLN characteristics rather than network properties to avoid binding it to any particular local area network. In fact, in the test configuration at BBN Cronus has been implemented on three different physical networks: Ethernet, Fibernet, and Pronet.

Cronus makes several basic assumptions about properties of the VLN, any of which may be implemented in software as necessary, although this may result in major reductions in network bandwidth. The assumptions are that:

- It is possible to send data packets between any two hosts on the VLN,
- The receiver must be able to determine the sender's identity,
- The maximum packet size is at least 1500 bytes, and
- There is a broadcast or multicast facility available on the network.

B.2.2. TCP/IP Protocols

Cronus uses TCP/IP to standardize communication between hosts on the VLN. These protocols multiplex incoming network packets and provide reliable communication over an unreliable physical network. In addition they provide the ability to communicate with hosts beyond the bounds of a single physical network. The Cronus Kernels use TCP to support reliable communication among themselves and Cronus processes may establish direct TCP links between each other.

TCP/IP was chosen largely because it is a standard and is available as a turn-key package for various COSes. On all of the systems in the test configuration but one it was possible to acquire an off the shelf TCP/IP. However, it was often the case that these implementations were unreliable and in some cases lacking appropriate functions to support their intended use in Cronus. Cronus uses the network communications facilities in a manner dissimilar to its use in standard higher level protocols such as *telnet* and *ftp*.

B.2.3. Cronus Kernel

The Cronus Kernel provides has three essential functions: local process manager to support Cronus client and manager processes, communication with local processes and with kernels on other hosts to support interprocess communication, and mechanisms to locate the manager responsible for a particular object to support object based IPC addressing for *invoke*. These are merged into a single component to facilitate data sharing and to provide a hard boundary between the secure and non-secure system components.

Cronus Kernels communicate with one another using the Cronus Peer to Peer protocol. Reliable delivery of datagrams is guaranteed through the use of TCP as the transport mechanism. The Peer to Peer Protocol includes a specification for establishing TCP links between Kernels and for closing them down. A provision is also included to send low effort datagrams and broadcast/multicast messages between Kernels using UDP datagrams.

The Kernels also provide basic process management services. These include process start up and removal functions and support for maintaining process identities and inheritances. These are implemented as Cronus operations on process objects.

The amount of system support required by Cronus is limited. This includes access to network services, interprocess communication (IPC) for communication between the kernel and Cronus processes, simple process support functions, and a function to generate Cronus Unique Numbers (UNOs) which are bit strings that are guaranteed to be unique over a Cronus system's life time. These are all described in some detail in the Cronus System Subsystem Specification.

B.2.4. Program Support Library

The program support library (PSL) consists of the subroutine interfaces to standard Cronus system calls and functions. It includes IPC primitives as well as the subroutine library for constructing and parsing messages (the MSL). In addition, various routines are included to support standard programmer requirements.

The idea behind the PSL is to make useful programming functions available to Cronus programmers on any COS environment. It is intended that the PSL should facilitate writing portable Cronus programs. On a particular COS, though, this may not be a concern, especially if a non-portable or unusual programing environment exists on the COS.

B.2.5. User Interface Programs

The nature of a particular COS determines what user interface programs should be implemented. If it is simply a service host with little user interaction only system maintenence programs are required. If it is a general purpose access point, a full complement of user programs should be provided including programs for file editing, directory listing, and status probing.

B.2.6. Manager Development Tools

Cronus is an object oriented system. Most conventional system services in Cronus are implemented as operations on system objects such as processes and files. So that application programmers can make use of the object model to define their own new object types a set of tools has been designed to simplify the tasks of defining new object types and of writing new manager programs to handle them.

The manager development tools consist of a collection of library functions, a manager specification language, and a code generator. All of these are oriented towards simplifying manager development in a C language environment. Currently the manager specifications are compiled on a machine running 4.2BSD Unix, but the generated code is portable to most C compilers without modification.

The manager devolopment library code includes functions for maintaining an object database, default object handling routines, and a multi-tasking facility to allow several concurrent tasks within a single process. Most of the functions are portable to any COS with a C development environment and a standard library interface. The tasking package, however, must be recoded for each target COS.

B.2.7. Existing Managers

There are several Cronus system services that are typically added to new hosts. These are described in detail in the Cronus System/Subsystem specification. Among the most commonly installed are the Cronus catalog and file managers, and the constituent operating system directory and file managers. Installing Cronus services extend the resource capacities of these services and, in the case of the catalog, improve the likelihood of surviving failures of other hosts. To facilitate this installation, the managers have been written in C, with special precautions to improve their portability. Installing the COS managers allow the new host's native directories and files to be accessed, maintained and administered from remote locations. These managers are also written in C, but in two parts: a machine independent part that implements the operation dispatching and general purpose portions of the operations, and a host dependent portion that uses native host system calls to access and modify the constituent directories and files.

Several other managers exist, both for the system and applications. Whenever possible, these are written to be transportable.

B.3. Integration Costs

There a number of factors that determine the cost in effort to integrate a new COS into Cronus. The single biggest question tends to be whether the code already written for other Cronus systems may be ported to the new system. Also critical is whether the COS already has a network interface and a TCP/IP driver. If the answer to these questions is favorable, experience has shown that the cost in man hours can be reduced by as much as ninety percent.

Most of the existing Cronus code has been written in the C programming language and, wherever possible, has been written to be portable. COS and host architecture dependencies are contained in several well defined modules. Because of this, it has been possible to port the same source code to all the systems in the Cronus test configuration. It should be noted, however, that Cronus is not dependent on C, and a Pascal version of the Cronus Kernel was implemented as part of a related contract effort.

TCP/IP and Ethernet were chosen as the bottom two layers in Cronus mainly because they were standards for which most COSes already have commercially available support. Experience has demonstrated that when these are "off the shelf" technologies, getting a Cronus Kernel running is greatly simplified.

The code supporting the higher levels of Cronus tends to have even fewer COS dependencies. Two areas, however, that have required special support code on each system include terminal/window interface routines and the multi-tasking facility used inside of the Cronus managers. It is difficult to assess the level of effort spent on the former since very few of the Cronus applications require special interactive interfaces. The multi-tasking facility, though, has been implemented on all the Cronus hosts and it has generally required between one and four man-weeks to write the machine specific code.

B.4. Implementation Issues

Cronus is typically installed as a collection of application programs that communicate using facilities built on the host's IPC and networking facilities. Some COS's already have very good facilities for supporting Cronus, while others are extremely inflexable and require significant modifications to the COS itself in order to support a Cronus implementation. The factors that can complicate an integration effort fall into four catagories: Processor Architecture, Network Interface, Programming Environment, and COS Problems. These are not always independent; e.g. a COS can restrict access to the network interface.

B.4.1. Processor Architecture

Although most existing Cronus code has been written in a high level language, the processor architecture characteristicly creates problems on each new integration. The design of Cronus does provide provisions for most standard processor incompatabilities, though. The Operation Protocol (OP) is used to encode and decode standard message data. By using abstract data types, OP masks architectural differences such as word size, byte ordering and data alignment restrictions. A set of functions known as the message structure library (MSL) is used to create and parse messages.

B.4.2. Network Interface

The ability to access the VLN is critical for the operation of Cronus. The Cronus Kernels communicate with one another using TCP links for reliable data transfers and using UDP datagrams for status probes. User programs may also establish TCP links with one another for large data transfers. In addition, the VLN is assumed to have a broadcast or multicast facility.

Since Cronus exercises both network hardware and software more than and in different ways from most applications, it tends to uncover bugs and problem areas. Some of the issues that have come up on the different systems in the test configuration include:

- Non-standard IP broadcast addresses,
- Incompatible schemes for mapping physical network addresses to IP host addresses,
- Insufficient buffering capability in hardware or software to handle standard network traffic,
- Buggy and non-standard UDP implementations,
- Software inability to handle maximum physical network buffer sizes,
- Software problems in dealing with Arpanet Standard Gateways, and
- Non-standard restrictions in using TCP port numbers.

Many of these problems were simply bugs with commercial software. For these problems, vendors varied from eager to totally uncooperative in coming up with fixes or work-arounds.

B.4.3. Programming Environment

The programming environment plays a key role in a COS integration. As with any large scale software implementation project, it is imperative to have a reliable and complete set of development tools available. This includes a C compiler, a linker, a debugger, and a librarian utility.

On several COSes the environment has been a source of difficulties. The C70 linker/librarian has trouble handling the number of object modules in the program support library. Also, the original cross compiler for the 68000 CMOS systems had a number of bugs which were only found through hours of source code/assembler output cross checking.

B.4.4. COS Problems

Cronus puts fairly high demands on resources of the target COS. Bare minimum requirements for a Cronus Kernel implementation include: facilities to add new system calls, or at minimum, an efficient, flexible intra-host IPC mechanisms to emulate a system call; a reliable stable storage interface; and some provision for performing asyncronous I/O requests to network channels. For a complete integration including user and manager programs it also requires a large process address space and it must be able to handle long lived processes in a graceful manner.

All of these requirements are generally available on a large machine operating system, and are increasingly available on many mini-computer and micro-computer bases systems. Current trends in micro-computers lead us to believe that the availability of these facilities will continue to grow. This was not true during our early development stages, and it proved to be a major stumbling block in the initial Cronus implementation effort on C70 UNIX.

B.5. Test Configuration Experience

This section describes some of our experiences porting Cronus to various COSes on the BBN test configuration. Each example starts with a discussion of the rational for adding the COS to Cronus and includes some discussion of the implementation issues and what goals were achieved.

B.5.1. SUN 4.2BSD

The SUN workstation is a high performance graphics oriented personal computer. It runs SUN's proprietary version of the Berkely 4.2 BSD UNIX operating system. The SUN was a particularly good candidate for integration into Cronus because of its graphics capabilities and because it was known to have a solid COS interface to TCP/IP.

Implementing the bottom levels of Cronus was fairly straight forward on the SUN. SUN's Ethernet interface satisfied the VLN requirements. The TCP/IP implementation was also acceptable; but there was a small incompatability with the other Cronus hosts on the net involving the IP network broadcast address. This was fixed by patching the UNIX kernel.

The Cronus Kernel was transported to the SUN without significant problems. The Kernel was implemented as a user process and relied on the 4.2BSD interprocess communication facility to communicate with user processes. A new UNIX system call was added to generate UIDs. This required some knowledge of the UNIX kernel.

SUN's C compiler and development environment proved to be adequate to port the PSL by rewriting the COS dependent modules to work under the 4.2 BSD environment. Certain problems did arise because of bugs and non-portable constructs in the code, but these were found and fixed as they were discovered.

Although the effort went smoothly, Cronus performance was somewhat of a disappointment on the SUNs. Because the interprocess communication facility on the SUN was significantly more expensive than on any of the other hosts on the configuration Cronus ran about 10-20% slower than the slowest of the others. Nonetheless, because of their availability SUN Unix became the main development environment for Cronus.

B.5.2. VMS

One of the first machines in the BBN test configuration was a VAX 750 running VMS. VMS was considered an important test of the portability efforts in Cronus, since it was the first Cronus COS that was completely unrelated to UNIX. It seemed likely, though, that the integration would not be too difficult since VMS was known to be well suited for building large subsystems.

The goal under VMS was to use as much off the shelf software/hardware as possible, so the initial efforts were to purchase a C compiler and a TCP Ethernet implementation for VMS. Both original C compiler and the original TCP/IP package that were chosen proved to be inadaquate and were later replaced as new producets entered the marketplace.

VMS had a fast efficient interprocess communication mechanism called mailboxes and it had many hooks for creating and monitoring processes. The Cronus Kernel VMS implementation makes use of these mechanisms as a user proccess. The Cronus Kernel was ported to VMS by writing new system dependent modules. The UNO generator was implemented as a VMS system service for efficiency reasons.

DEC's C compiler and development environment were very good and presented few problems in porting the PSL or the manager development tools. There were several minor incompatabilities with other COS environments (for example the VMS linker was not case sensitive) but these were all easily handled.

B.5.3. CMOS

It was decided early in the Cronus design effort that if Cronus could be implemented on a bare machine with a minimal or no COS, it might be possible to get improved Cronus performance on low cost hardware. These systems are called Generic Computing Elements (GCEs). A 68000 based micro processor GCE was configured, and CMOS, a small real time operating system, was chosen to provide the initial support for low level system functions.

The problems associated with CMOS were quite a bit different than on any of the other COSes. It was necessary to write device drivers for the Ethernet interface and the disk controller. Since no implementation was readily available, a complete TCP/IP package was implemented.

Given the necessary lower level tools it was possible to port the Cronus Kernel to CMOS with relatively few modifications. As it was assumed that the GCEs would be dedicated service hosts, no support for dynamic process creation or removal was implemented. The rest of the kernel functions, though, were all ported.

The PSL and some components of the manager development library were ported to the CMOS environment. The first generation GCEs had some significant physical memory limitations and various hardware problems, but it was possible to run several services on them. They were used as file servers and terminal multiplexing access points to Cronus. Currently they are being replaced by a next generation GCE which will have many more capabilities.

B.6. Conclusions

Overall the various COS integration efforts in BBN's configuration have proceeded smoothly. All of the target machines have been successfully integrated at least sufficiently to run certain object managers. Most importantly, this has been done without altering the design of Cronus and without introducing any incompatabilities or limitations into the system. Experience has shown that the design of Cronus is host independent in a practical sense as well as a theoretical one.

C. Ethernet Experience

Cronus, A Distributed Operating System: Experiences in Operating An Ethernet

Richard Sands

BBN Laboratories Incorporated
10 Moulton Street
Cambridge, Massachusetts 02238

C.1. Introduction

For the past three years, the Cronus project has owned and operated an Ethernet, used in the development of the Cronus distributed system testbed for RADC [1, 2]. This network currently has 25 hosts and 2 gateways, including a wide variety of hosts, operating systems, and Ethernet hardware. The network currently includes:

Processors		
VAX	MASSCOMP	68000 GCE
C/70	Symbolics	LSI-11 gateway
SUN	Butterfly	

Operating Systems		
4.2 BSD Unix	BBN CMOS	gateway software
BBN OS (V7 Unix)	MASSCOMP Unix	Symbolics Lisp
VMS	BBN Chrysalis	

Ethernet Interfaces		
3Com Unibus	Interlan Qbus	Excelan Multibus
3Com Multibus	DEC DEUNA	Symbolics
Interlan Unibus	BBN Mieni/Interlan NM10	

Tranceivers		
3Com	Interlan	DEC

Network Protocols		
ARP/IP/TCP	ARP/Chaos	

As can be seen, there is a very wide variety of hosts. It has been our experience that all of these disparate hosts can be made to coexist and intercommunicate on the same Ethernet. However, this intercommunication capability has been achieved only after considerable headache and work. The purpose of this note is to capture some of the experience that has been accumulated in managing the Cronus Ethernet, and to indicate the types of problems which can be expected in the management of similar Ethernets. Though several high-level protocols coexist on the Cronus Ethernet, this note will be primarily concerned with describing our experiences with IP [3], the base-level Internet datagram protocol for the DOD.

The IEEE has developed a set of local-area network (LAN) standards designated IEEE 802. This family of protocols specifies physical and data link layers for two types of LANs, bus and token ring [4, 5, 6, 7]. The bus protocol (IEEE 802.2/802.3 [4, 7]) is based on the Ethernet Version 2 standard [8]. The Ethernet Physical Layer and IEEE 802.3 are fully compatable, but the Ethernet Data Link Layer and IEEE 802.2/802.3 Data Link Layer are incompatable. The Cronus Ethernet conforms to the Ethernet Data Link Layer, rather than the IEEE 802.2/802.3 Data Link Layer.

C.2. The Ethernet Coax and Physical Network Layout

The Cronus Ethernet did not spring into its current configuration in one step. It has slowly evolved from a small experimental network with only a few hosts to its current size. Initially, we had no experience in setting up an Ethernet, and in hindsight, we made a number of mistakes in the physical layout of the network.

The Ethernet standard [8] requires that the distances between taps on the cable be a multiple of 2.5 meters. Ethernet coax is required to have index marks every 2.5 meters to facilitate in meeting this requirement. Our original batch of coaxial cable did not have 2.5 meter index marks, making it very difficult to cut the cable into legal lengths. We thus have several pieces of cable in our network which we are not sure of their length, and which in any case are of a non-standard length. Because our network is not technically in compliance with published specifications, we can never rule out the possibility that the inability of two hosts on the network to communicate is due to a faulty network backbone. In addition to out-of-spec inter-tap cable lengths, there is a restrictive requirement that inter-tap distances be one of several odd lengths if the Ethernet is to be composed of cable pieces from different manufacturers or different batches, to eliminate the effects of signal reflections at points where the impedance changes slightly. We have not followed this specification either, which has inhibited us from installing some cable pieces from different manufacturers (such as TEFLON pieces). To eliminate the possibility that difficulties are caused by an out-of-spec Ethernet, be sure to scrupulously follow the IEEE 802.3 / Ethernet 2.0 specifications as to cable type, allowable lengths, insulation, termination, etc.

Ethernet cable may be purchased either in bulk, or in precut, connectorized pieces. Buying in bulk is less expensive, and more reliable, since it has been our experience that connectorized, precut pieces of Ethernet tend to be shoddily assembled. Buying in bulk has the additional benefit that the cable is guaranteed to be from the same batch. An Ethernet made entirely from a single batch of cable can have less restrictive requirements for tap placement, and should be less susceptable to problems caused by spurious reflections at impedance mismatch points. For maximum economy and reliability, buy Ethernet coax in bulk, preferably enough for anticipated growth of the net, to avoid multiple batches and connector reliability problems. 500 meters is the maximum segment size, and is thus enough per batch for any eventuality.

If there is a problem with the Ethernet coax (shorts, open connections, grounded connectors, etc), debugging the network will involve a binary search for the problem cable piece by successively breaking the network at connectors and testing for 50 ohm impedance on either side of the break with a multimeter. Thus, it is neccessary to know the locations of all cable pieces and connectors, and to know where each end of a cable piece is. To facilitate this, keep an accurate map of the network with the room numbers of all connectors and ends of cable pieces shown. This map MUST be kept accurate! Also, be sure to label each end of a cable piece with the location of the other end.

Once an Ethernet has been installed, management of the physical network consists of fixing problems with the cable or connectors, and insuring that no additions or modifications to the net cause any segment to exceed the maximum allowed length of 500 meters. Careful measurement and accounting of cable lengths are required so that the lengths of segments are always known. If the network must grow longer than 500 meters, a repeater may be used between cable segments, with up to 1500 meters total length between any two taps on the network.

C.3. Ethernet Tranceivers

It has been our experience that all tranceivers we've tried have been able to intercommunicate with each other on an Ethernet. We have experienced almost no problems with tranceivers, with only one failing so far during the lifetime of the net. In practice, they are all equivalent, and any tranceiver should do just fine.

There are two main types of tranceivers. The non-intrusive "vampire" taps (DEC, Interlan) connect to the coax through a probe that is inserted into a carefully drilled hole that reaches through the various layers of the cable almost to the core. A special tool kit is required to install these taps, but the taps are cheaper, and can be installed (carefully) on a live network with no disruption.

The connectorized taps (3Com) have female coax connectors on either end, and are attached to the network by screwing on cable pieces with male connectors. They are more expensive than the vampires.

All tranceivers are compatable with all Ethernet interfaces with one exception; if the Ethernet interface conforms to the IEEE 802.2 specification, a tranceiver that also complies with this specification is required. The only difference between tranceivers that comply with IEEE 802.2, and those that do not, is the capability of the IEEE tranceivers to loop back signals just before they reach the coax, to allow for complete testing of the interface by operating system software before enabling the interface for network I/O. Be sure to use IEEE 802.2 tranceivers with interfaces that require them. Otherwise, any tranceiver will do.

C.4. Ethernet Interfaces

The Cronus project has had a lot of experience with various vendors and their Ethernet hardware. A list of interfaces and associated comments follows:

3Com

This product provides a relatively low-capacity Ethernet interface. It is low-capacity primarily because it is not a DMA device, and because it requires that the CPU handle most of the details of the Ethernet data-link protocol such as the collision handling algorithm, including computing binary exponential backoff times. It thus consumes considerable CPU resources just in moving frames onto and off of the interface's on-board memory.

The requirement that the CPU perform most of the Ethernet protocol means that device drivers for this interface are relatively complicated. An incorrect, misbehaved implementation can also wreak various amounts of havoc with communication on the net. On the other hand, because the capabilities of the interface are so closely tied to the device driver, it is possible to provide a very capable interface, including such features as arbitrary numbers of multicast addresses recognized, etc.

Interlan

All Interlan interfaces use a daughter board (called the NM10A) which actually implements the Ethernet data-link protocol. Interfaces to specific processor busses are in essence adapter cards which interface the NM10A to the specific system. The NM10A has substantial on-board buffering for incoming packets, lessening the chances that packets will be dropped. Interlan interfaces use DMA transfers to move data into and out of main memory, and thus offer higher performance with less CPU overhead than the 3Com interfaces. The Interlan boards offer self-test diagnostics, and are able to recognize up to 8 multicast addresses in hardware. I have heard that Interlan also offers an intellegent interface board that implements ARP/IP/TCP in firmware. This technique holds the promise of off-loading a significant amount of operating system overhead from system CPUs, and placing it on specialized hardware.

Mieni/Interlan NM10A

The Mieni is a custom interface designed and implemented in the early Ethernet days to connect C/70s to the Ethernet. There are only four in existence, and maintenence is quite difficult and expensive. It uses the Interlan NM10A daughter board to implement the Ethernet protocol, and interfaces this board to the MBB processor.

Building and debugging the Mieni board took a very long time, and was quite expensive. In general, custom Ethernet interfaces are probably a bad idea, unless it is very important to put a machine on the Ethernet, and there are no commercial interfaces available. Possible alternatives to custom Ethernet interfaces might include a front-end processor with some form of high-speed communication with the host.

DEC DEUNA

This IEEE 802.2 interface requires an IEEE 802.2 tranceiver, and to be safe, should probably be used only with a DEC tranceiver. It is a DMA device, and seems to perform well, with no unusual problems. These interfaces have been used by the BBN Division 4 Computer Facility to connect VAX timersharing systems to the Ethernet; the Cronus project itself has no experience with them.

Excelan

This company is relatively new to the Ethernet interface market, and seems to offer fairly high-performance interfaces that may be configured as intelligent interfaces implementing ARP/IP/TCP in firmware. These boards are downloaded with their protocol software from the host CPU. and thus may be customized to perform any desired high-level protocol. We have only recently begun to use these devices on the Cronus project to provide Ethernet access for the MassComp GCE. Offloading the network protocol processing to the network interface board might provide performance improvements when compared with the existing general purpose, single processor implementation.

Sometimes, the choice of Ethernet interface will be predetermined, if there is only one manufacturer for the machine, or if the Ethernet interface is bundled in with the rest of the system's hardware. However, if there is a choice possible, buy Ethernet interfaces from reputable companies, that perform at least the Ethernet data-link protocol in hardware. New intelligent interfaces hold the promise of substantially reducing the overhead of connecting to a network, but we do not have sufficient experience yet to confirm the performance claims.

C.5. The Address Resolution Protocol

The Ethernet data-link protocol provides for addressing of individual hardware stations on the locally connected physical cable, of groups of stations (multicast), and of all stations (broadcast). Higher-level protocols such as IP also provide for addressing. There must be a technique for mapping higher-level protocol addresses into 48 bit Ethernet station addresses. This may be done in several ways, the simplest of which is a static table. For small networks serving a dedicated role with a fixed number of well-known hosts, a static table would be a sufficient mechanism for address mappings. For the more typical service network, a dynamic discovery procedure is preferable.

The Address Resolution Protocol, (ARP, [9]) is a standard technique for dynamically mapping higher-level protocol addresses to the Ethernet's (or any other broadcast network's) station addresses. Hosts on the Cronus Ethernet must correctly implement ARP to communicate with other hosts on the net. Greatly simplified, the basis for the protocol is that a host that must identify the Ethernet station address corresponding to a higher-level protocol address broadcasts an ARP request, and receives a response with the required address translation from the destination host. Translations are stored in a table, so that future packets bound for the destination are address mapped without resorting to ARP. As an optimization, all ARP implementations on an Ethernet that receive a broadcast request update their table entries for the source of the request, making it unneccessary for these hosts to use ARP to translate this host's address in the future.

Though the basic idea of ARP is quite simple, in practice, the protocol poses some tricky implementation issues. In addition, the protocol has some problems intrinsic to its algorithm that can cause trouble.

One common ARP implementation error is improper handling of ARP requests for address mappings of higher-level protocols not supported by a host. A host must silently throw away any ARP request for translation of a higher-level protocol which it doesn't implement. Printing error messages, saving the sender's mapping in the translation table, or any other action is incorrect. This bug seems to crop up in implementations that were developed in an environment that has only IP as a higher-level protocol on top of raw Ethernet. ARP can enable multiple logical networks using different higher-level protocols to coexist on an Ethernet. This problem first appeared when the CHAOS protocol LISP machines were first installed on the Ethernet.

Another problem is intrinsic to ARP itself, and is identified as a problem in [9]. This problem occurs when an Ethernet station address is reassigned to a different host (usually by swapping Ethernet controllers for testing purposes). Hosts which have already acquired the translation for the host whose address has just been reassigned retain the outdated mapping until the reassigned host broadcasts an ARP packet with the new address, updating the translation tables. Until this update occurs, packets cannot be routed to the reassigned host from hosts which have acquired the old mapping. The designers of ARP assumes that reassignment of Ethernet addresses is quite uncommon so that the problem can be ignored. It has been our experience that one of the best ways to debug Ethernet hardware is by swapping identical parts for suspected problem components, and seeing if the problem goes away. This type of substitution debugging is to some extent stymied by the inability of ARP to cope with changing hardware addresses for hosts.

This problem can be corrected in several ways. One way is to age entries in the address translation table, so that communication failure doesn't last until a reboot of the system with faulty table entries. Another technique for correcting this problem is a method for manually purging bad entries from the translation table, or clearing the table outright. This second solution is difficult for a non-technical person to use, and would likely have to be performed by system maintainers. Other techniques that can help with this problem include having higher-level protocols purge translation table entries of hosts for which a connection cannot be opened, or having newly booted hosts broadcast an ARP request for themselves, thus resetting the translation tables of all hosts receiving the broadcast.

C.6. Transmitting IP Datagrams On An Ethernet

Most implementations of IP on an Ethernet conform to [10]. A standard for encapsulating IP datagrams on on IEEE 802.2/802.3 network has also been defined, in [11]. The Cronus Ethernet uses the Ethernet encapsulation. The 4.2 BSD implementation for the VAX uses a non-standard encapsulation of IP in Ethernet frames called a 'trailer encapsulation', for performance reasons [12]. Use of trailers on the Cronus Ethernet has been eliminated, by patches and bootstrap parameterization on affected systems. Though performance may be increased somewhat between 4.2 systems using trailers, the inability to communicate with other systems seems too great a price to bear.

C.7. Internet Broadcast

The Ethernet is inherently a broadcast medium, since all stations on the Ethernet "hear" all transmissions. The Ethernet controller hardware on each host discards all packets not intended for that host, as determined by the Ethernet destination station address in the Ethernet header of the packets. There is a single assigned Ethernet address, consisting of 48 bits all ones, that is received by all Ethernet controllers regardless of their station address. This special address, called the *Ethernet broadcast address*, may be used to direct a message to all hosts on an Ethernet simultaneously. Such a facility is quite useful for many applications.

As with specific station addresses, there must be a mapping from some higher-level protocol address to the Ethernet broadcast address. ARP is not needed to translate this address however, because the mapping is standard and does not change. A standard has been specified in [10, 11], and reiterated in [13], in which the host part of the 32-bit IP address on that net is all binary ones. For class A networks, the IP broadcast address on that network is of the form X.255.255.255, for class B networks, X.X.255.255, and for class C networks, X.X.X.255. All broadcast IP packets on an Ethernet must have the IP broadcast address in their IP destination address fields.

The Ethernet controller hardware takes care of recognizing the Ethernet broadcast address, and passing packets received from that address to higher-level protocols. IP must then recognize that it should receive packets addressed to the IP broadcast address, rather than flagging such packets as IP address errors. IP then hands the packet up to still higher-level protocols. If a host recognizes a non-standard IP broadcast address, it will not be able to broadcast to the hosts on the net meeting the standard, because these hosts will discard the non-standard host's broadcasts as not addressed to them. Conversely, the non-standard host will not receive broadcasts from standard hosts for the same reason.

Unfortunately, 4.2 BSD Unix and its derivatives, the most common type of network software on the Cronus Ethernet, uses a non-standard IP broadcast address with the host part of the IP address zero, rather than all ones. These hosts cannot communicate via broadcasts with the hosts on the Cronus Ethernet that observe the standard. To make matters worse, the 4.2 BSD implementation acts as a gateway for packets it receives that are not addressed to it. Using a simplistic and faulty algorithm for gatewaying packets, the 4.2 BSD implementation assumes that the broadcast packets it receives that do not use its non-standard broadcast address are actually packets to be gatewayed to another network. Not noticing that the source and destination networks are the same, the software attempts to forward the packet back onto the Cronus Ethernet. Luckily, the ARP request to translate the correct IP broadcast address is not responded to, and the forwarding fails. This problem results in a large number of unneccessary ARP request packets on the network, however.

It has been necessary to either replace or patch the 4.2 BSD Unix network software, to implement the standard IP broadcast address on these systems. On VAX systems, the BBN network software has replaced the 4.2 BSD software, providing the correct address. On SUN workstations, a special version of the Unix kernel that has been patched with the correct broadcast address has been produced, and now runs on SUN workstations on the Cronus network. The Wollongong Group TCP/IP implementation for VMS, a 4.2 BSD derivative originally used the non-standard address. In response to our requests, they have provided a version which can be configured to use either the IP or Berkeley 4.2 standard broadcast address.

To partially cope with the problems of the non-standard IP broadcast address, currently on the Cronus Ethernet, hosts that are able to will accept broadcast packets received with the non-standard address, but will only use the standard address for their own broadcasts. It is hoped that soon, no non-standard implementations will be on the Cronus Ethernet, and we can revert to having a single, correct IP broadcast address known and used network-wide.

C.8. Internet Multicast

The Ethernet also supports a directed broadcast, or *multicast* facility, in which a subset of the hosts on an Ethernet receive packets addressed to a special station address taken out of a reserved set of addresses. Though this facility would prove quite useful for certain applications, it is currently not used on the Cronus Ethernet because of anticipated operational issues.

Different Ethernet controllers offer varying amounts of hardware support for recognizing multicast addresses, and thus, it is a design issue how to map an unlimited IP multicast facility onto the limited support provided by the hardware. One technique for providing such a multicast facility is described in [14].

C.9. The IP Layer and Gateways

Local-area networks such as the Ethernet gain considerable utility by being tied into the Internet through a gateway. The Cronus Ethernet has a three-legged gateway connecting it to the ARPANET and the BBN Fibernet.

When a host on an Ethernet communicates through a gateway with a host on the ARPANET, the mismatch of maximum packet sizes between the two networks causes large Ethernet packets to be fragmented at the gateway before being forwarded onto the ARPANET. Fragmentation can cause a substantial reduction of throughput, or communication failure between two hosts, so it may be advisable for a host to use smaller packets when communicating with another host not on the local Ethernet. These losses from fragmentation are due to a combination of the speed mismatches between the networks, the unreliable transmission of fragments, and the 8 packet in transit limit between two hosts on the ARPANET. These factors combine to make fragmentation, originally designed as a technique for facilitating such inter-network communication, a very questionable technique. In addition to packet fragmentation, the mismatch of maximum packet sizes causes a buffer allocation problem in the gateway, which is an LSI-11 with only a limited amount of memory. The gateway can either allocate several different sizes of buffers for the different sized packets it can receive from its networks, or allocate all buffers large enough for the largest packets it can receive from any network. In the first case, memory fragmentation can result in inefficient use of buffer memory, and in the latter case, large buffers used for small packets also results in inefficiencies.

Since fragmentation is undesirable, and gateways are strapped for memory as it is, there is an incentive for administratively reducing the size of packets on an Ethernet to match the packet size of networks on the far side of a gateway. This approach has the benefit that inter-network communication becomes more reliable when the packets take an unusual or circuitous route to their destinations. The drawback to this approach is that there is a published standard in RFC-894 which requires gateways to accept full-length packets, and fragment them if neccessary. In a very herogeneous environment such as the Cronus Ethernet, there will be (and are) hosts for which the maximum transmission unit for the Ethernet cannot be conveniently made smaller. These are hosts that use vendor-supplied network implementations for which we have no source code, and which are not dynamically configurable. These hosts are for the most part unable to communicate off of the local network. In addition, since the per-packet obverhead of TCP/IP on most hosts seems fairly constant over a range of packet sizes, actual throughput between hosts is almost linearly related to the packet size in use. Further benchmarking is needed to verify this conclusion, however.

Ultimately, the only solution to these problems is enhanced inter-network protocols that deal directly with them. In the meantime, it has been our experience that the communication problems caused by use of full-sized packets on the Cronus Ethernet are not severe enough to justify administratively reducing the packet size on the network, with consequent problems for some hosts, and reduced performance in intra-network communication.

A discussion of these and other related issues may be found in [15]. Though there is currently no evidence that the Cronus gateway is under an unduly heavy load, or is dropping a disporportionate number of packets, continued care has been necessary to insure that patterns of usage do not cause the gateway performance to degrade.

C.10. Network Applications and Ethernet Broadcasts

The broadcast facility of the Ethernet has proven to be very useful in a number of applications, ranging from simple statistics and 'whos on the machine' daemons, to object location mechanisms for the Cronus Distributed Operating System.

Several characteristics of broadcasts limit their usefulness. These are their unreliable transmission, overhead on all hosts on the net, and their limitation to hosts co-located on a single Ethernet. All of these problems have been solved by Cronus, or at least minimized.

Unreliable transmission of broadcast packets, usually via UDP, has not been a problem, for the most part. The Ethernet is a surprisingly reliable transmission medium, and even with no end-to-end acknowledgements and retransmissions available, broadcasts are ordinarily received correctly by all hosts on the net. Broadcasts are lost usually from overloaded Ethernet interfaces on individual hosts, rather than from garbling of the message itself on the Ethernet. Our current approach to managing the unreliability of broadcasts has been to design our applications to be tolerant of some loss of broadcasts, rather than to implement a costly and complicated simulation of reliable broadcast using other reliable transmission techniques.

The overhead of broadcast packets on hosts that are not interested in them (ie. no program waiting on the UDP socket) is a problem that has not been fully solved on the Cronus Ethernet. The incorrect broadcast address used by some SUN workstations causing a flood of broadcasted ARP requests results in a substantial increase in the number of broadcasts on the Cronus Ethernet. Once this problem has been resolved, it will be time to do some performance benchmarks to determine exactly the penalty for too many broadcast packets on the net.

The limitation of using broadcasts only within the boundaries of a single local network has been eliminated using a Cronus component called the *broadcast repeater*. This component is described in [16]. Briefly, the broadcast repeater consists of a pair of processes on hosts residing on different networks. These processes are linked by a TCP connection, and broadcast packets are relayed over the TCP connection for rebroadcast on the non-originating network. Packet filters insure that only the packets that are truly important get rebroadcast.

C.11. Telnet and Ethernet Terminal Concentrators

In the course of bringing up TCP/IP on standalone servers based on the 68000 CPU, called GCEs, the Cronus project produced a very effective Ethernet terminal concentrator as a test of the protocol implementation. This terminal concentrator provides telnet access to any host on the Internet, provides up to 4 active connections per user, and services 8 users simultaneously. It has excellent performance, with no discernable delay imposed by the implementation, or the network communication. In daily use for nearly a year, this terminal concentrator has become the preferred network access mechanism for a number of Cronus project members.

One important lesson learned from use of this terminal concentrator is that multiple connections per user is such a useful feature that it should be a required feature for any terminal concentrator supplied by an outside vendor

C.12. Network Administration

A forum for the users of a network to find out about changes and hold technical discussions seems to be desirable in administering an Ethernet. Coordinating the installation of new hosts, changes to the network backbone, and testing of new network implementations has been greatly facilitated by a mailing list, on which interested parties can find out about administrative schedules.

C.13. Network Monitoring and Control

The different host types in use on the Cronus Ethernet offer varying amounts of monitoring and control over their use of the Ethernet. Unix systems offer the **netstat** command, which details the higher-level protocol connections currently established with other systems. C/70 systems have in addition the **etherstat** command, which presents Ethernet usage statistics such as the collision rate, the transmit and receive rates for both specifically addressed and broadcast packets, and the rates of various detectable transmission errors such as frame misalignment and CRC errors. Many statistics that might prove useful for tuning performance are not available, however, such as the IP fragment receive rate. The gateway fragmentation rate may be had from gateway statistics collected by the gateway maintainers. Many other useful statistics are available from this source as well. Address translation tables may be displayed on a number of systems, revealing the causes of certain problems with ARP.

When debugging network communication problems, packet printers are quite useful. A packet printer is a program that eavesdrops on the network, and prints the protocol fields and data from packets selected by user-specified filters in an easy to read, formatted style. Such programs often reveal the causes of network failures when no other technique exists for tracking down the problem. Care must be taken in allowing people to run packet printers however, since data in Ethernet packets is not encrypted, and peoples' passwords and the like are transmitted in the clear over the network constantly. The Cronus project has two packet printers, one that runs on the C/70 and is able to display most packets, with the exception of ARP requests and replies. Another packet printer runs standalone on the GCE, and can print any packet on the net.

The various hosts on the Cronus Ethernet provide different degrees of control over the address translation tables used by ARP. Control over these tables is a useful feature, since it can eliminate the need to reboot a system to clear bad translations, and can make the choice of broadcast address dynamic. In addition, the C/70s and GCEs have the ability to register multicast addresses, which in conjunction with the table manipulation commands available, make Ethernet multicast possible on these hosts.

C.14. Broadcast Networks and Misbehaved Hosts

On a broadcast network such as the Ethernet, misbehaved hosts can cause substantial problems for other hosts on the network, and even cause complete communication failures or host crashes. An example of this type of dramatic network failure occurred when a faulty Unix kernel was installed on a SUN workstation during a Cronus demo. The faulty kernel had been modified to map the standard Internet broadcast address to the Ethernet broadcast address, but had not been modified to recognize incoming packets addressed to the standard Internet broadcast address as broadcast packets. Both modifications are neccessary for the kernel to use the correct broadcast address. This partial modification caused the SUN workstation to assume that received broadcast packets using the standard broadcast address were in fact not meant to be received locally. The 4.2 BSD network implementation will act as a gateway for packets it receives that were not destined for the local host. Because the kernel had been modified to know the address translation for the standard broadcast address, the system did not attempt to get the address translation using ARP, and fail. Instead, it successfully rebroadcast the packet back onto the Cronus Ethernet, heard its own rebroadcast, and rebroadcast again, etc. This caused a continuous stream of identical broadcast packets to be emitted from the SUN, nearly bringing down many hosts on the network, and at any rate, fouling Cronus communication up completely.

New network implementations, or changes made to existing implementations may often have bugs in them that can directly affect the other hosts on the network. Some controlled means for introducing such new implementations must be used, if unfortunate failures at inopportune times are to be avoided. On a full service network, such considerations would be even more important.

C.15. Performance

There has been a lack of significant benchmarking on the Cronus Ethernet to back up claims such as that IP fragmentation at the gateway is acceptable, if the alternative is decreased performance and trouble for unconfigurable hosts. Some performance information is available in the form of day to day experience with the network, however. In general, the 10 megabit/sec. bandwidth of the Ethernet has barely been scratched by the collection of 30-some hosts, including some quite powerful systems, on the Cronus Ethernet. We have found that an Ethernet collision is quite a rare event, indicating that the fears people have of severe deterioration of performance under current loads are unfounded. It would be impractical to actually have as many hosts on a single Ethernet segment as it would take to significantly overload it, because of the 500 meter length limit on a segment, and the realities of building design, computer sizes and costs, etc. using current technology.

Even though the Ethernet cannot be easily overloaded, there are two important performance characteristics that should be considered; transmission delay, and end-to-end throughput. Because of the high bandwidth of the network, most delay is incurred in the host operating system software, both transmitting and receiving the packet. Most systems are able to transmit or receive a packet somewhat slower than they would be able to transfer a similar amount of data to or from a disk. Thus, the end to end delay is small, but still not negligable. Throughput is likewise limited by the operating systems' ability to quickly move data through the network protocol software. Thus, most systems have not reached the performance level necessary for them to be primarily bound by disk bandwidth.

C.16. Conclusion: The Importance of Standards

With the number and variety of hosts on the Cronus Ethernet, it is surprising that for the most part, intercommunication among all the systems is quite effective and reliable. To a large extent this compatability is a direct result of the close adherance to published standards that most implementations have followed. In retrospect, nearly all of the problems we have had in the course of building and maintaining this network have been the result of some violation of a published standard that other hosts on the network have followed. Though at times, standards have not been far-reaching enough, or have proven ineffective at solving real-world problems, or have even introduced problems that did not exist before the standard was adopted, interoperability among diverse host types would not be possible without them.

No matter how distasteful some standards may be to implement, or what problems they might cause, the anarchy that non-standard implementations cause cannot be tolerated. When a standard is too onerous to implement, consumes too many resources, or causes problems not foreseen when the standard was designed, the correct approach to solving the problems is to revise the standard, and implement the revisions. This process can take a long time, and may not always provide an optimal solution, but special modifications or ad-hoc changes to standards will almost certainly insure that interoperability cannot be achieved.

Overall, our experience with the Ethernet has been a positive one. When the Ethernet was chosen as the local-area network technology on which Cronus would be implemented, it had not yet evolved into the de-facto industry standard that it is today. In retrospect, it is fortunate that the Cronus project chose the 'winner' of the local-area network competition, since the wide range of available products and implementations make the task of porting Cronus to other architectures and systems much easier.

C.17. References

[1] Schantz, R., and Thomas, R. **"CRONUS, A Distributed Operating System: Functional Definition and System Concept"**. Report 5879, Bolt, Beranek, and Newman, RADC-TR-88-132, Vol II dated June 1988.

[2] Schantz, R., et. al. **"CRONUS, A Distributed Operating System: Revised SYSTEM/SUBSYSTEM Specification"**. Report 5884, Bolt, Beranek and Newman, RADC-TR-88-132, Vol I dated June 1988.

[3] Postel, J. **"Internet Protocol"**. RFC-791, USC/Information Sciences Institute, September 1981.

[4] The Institute of Electronics and Electronics Engineers, Inc. **"IEEE Standards for Local Area Networks: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications"**. The Institute of Electronics and Electronics Engineers, Inc., New York, New York, 1985.

[5] The Institute of Electronics and Electronics Engineers, Inc. **"IEEE Standards for Local Area Networks: Token-Passing Bus Access Method and Physical Layer Specifications"**. The Institute of Electronics and Electronics Engineers, Inc., New York, New York, 1985.

[6] The Institute of Electronics and Electronics Engineers, Inc. **"IEEE Standards for Local Area Networks: Token Ring Access Method and Physical Layer Specifications"**. The Institute of Electronics and Electronics Engineers, Inc., New York, New York, 1985.

[7] The Institute of Electronics and Electronics Engineers, Inc. **"IEEE Standards for Local Area Networks: Logical Link Control"**. The Institute of Electronics and

Electronics Engineers, Inc., New York, New York, 1985.

[8] **"The Ethernet, Physical and Data Link Layer Specifications, Version 2.0".** Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, 1982.

[9] Plummer, D. **"An Ethernet Address Resolution Protocol".** RFC-826, Symbolics Cambridge Research Center, November 1982.

[10] Hornig, C. **"A Standard for the Transmission of IP Datagrams over Ethernet Networks".** RFC-894, Symbolics Cambridge Research Center, April 1984.

[11] Winston, I. **"Two Methods for the Transmission of IP Datagrams Over IEEE 802.3 Networks".** RFC-948, University of Pennsylvania, July 1985.

[12] Leffler, S., and Karels, M. **"Trailer Encapsulations".** RFC-893, University of California at Berkeley, April 1984.

[13] Reynolds, J., and Postel, J. **"Assigned Numbers".** RFC-943, USC/Information Sciences Institute, April 1985.

[14] MacGregor, W., and Tappan, D. **"The Cronus Virtual Local Network".** DOS Note 26, The Cronus Project, Bolt, Beranek, and Newman, August 1982

[15] Postel, J. **"The TCP Maximum Segment Size Option and Related Topics".** RFC-879, USC/Information Sciences Institute, November 1983.

[16] Lebowitz, K., and Mankins, D. **"Multi-Network Broadcasting Within the Internet".** RFC-947, Bolt, Beranek and Newman, July 1985.

D. Bibliography

Cronus System/Subsystem Specification. BBN Laboratories, Technical Report 5884. June 1982, RADC-TR-88-132, Vol I dated June 1988.

J. Berets, R. Schantz, **C2 System Internet Experiment: Interim Technical Report.** BBN Laboratories, Technical Report No. 5942.

J. Berets, R. Mucci, R. Schantz, **Cronus: A Testbed for Developing Distributed Systems.** 1985 IEEE Military Communications Conference. October 1985.

M. Dean, R. Gurwitz, R. Schantz, **Programming Support in the Cronus Distributed Operating System.** Sixth International Conference on Distributed Computing Systems. May 1986.

R. Sands, K. Schroder, **Cronus User's Manual.** BBN Laboratories, Technical Report 6180. February 1986.

R. Sands, K. Schroder, **Cronus Program Maintenence Manual.** BBN Laboratories, Technical Report 6181. February 1986.

R. Sands, K. Schroder, **Cronus Operator's Manual.** BBN Laboratories, Technical Report 6182. February 1986.

R. Schantz, E. Burke, S. Geyer, M. Hoffman, A. Lake, K. Pogran, D. Tappan, R. Thomas, S. Toner, W. MacGregor, **Cronus, A Distributed Operating System: Interim Technical Report No. 1.** BBN Laboratories, Technical Report No. 5086. July 1982.

R. Schantz, B. Woznick, G. Bono, E. Burke, S. Geyer, M. Hoffman, W. MacGregor, R. Sands, R. Thomas, S. Toner, **Cronus, A Distributed Operating System: Interim Technical Report No. 2.** BBN Laboratories, Technical Report No. 5261. February 1983.

R. Schantz, M. Barrow, G. Bono, M. Dean, M. Hoffman, R. Sands, R. Schantz, R. Thomas, B. Woznick, **Cronus, A Distributed Operating System: Interim Technical Report No. 3.** BBN Laboratories, Technical Report No. 5646. May 1984.

R. Schantz, R. Thomas, R. Gurwitz, M. Barrow, G. Bono, M. Dean, K. Lebowitz, K. Schroder, R. Sands, **Cronus, A Distirubted Operating System: Phase 1 Final Report (Interim Technical Report No. 4).** BBN Laboratories, Technical Report No. 5885. January 1985.

R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lebowitz, R. Sands, **Cronus, A Distributed Operating System: Interim Technical Report No. 5.** BBN Laboratories, Technical Report No. 5991. RADC-TR-88-132, Vol III dated June 1988.

R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lam, K. Lebowitz, S. Lipson, P. Neves, R. Sands, **Cronus, A Distributed Operating System: Cronus DOS Implementation, Final Report (Interim Technical Report No. 6).** BBN Laboratories, Technical Report No. 5885. February 1986.

R. Schantz, R. Thomas, G. Bono, **The Architecture of the Cronus Distributed Operating System.** Sixth International Conference on Distributed Computing Systems. May 1986.

Richard E. Schantz, Robert H. Thomas, **Cronus Functional Definition and System Concept.** BBN Laboratories, Technical Report 5879. RADC-TR-88-132, Vol II dated June 1988.

MISSION
of
Rome Air Development Center

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