RADC-TR-88-159, Vol II (of two)
Final Technical Report
August 1988

AD-A203 845

RESEARCH IN DISTRIBUTED PERSONAL COMPUTER-BASED INFORMATION SYSTEMS
Semi-Annual Technical Report No. 5

BBN Laboratories Inc.

DTIC
ELECTE
1 0 FEB 1989
E

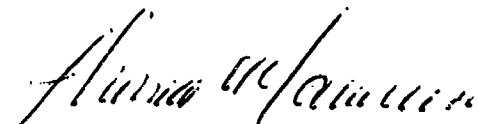ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

89 2 9 180

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.
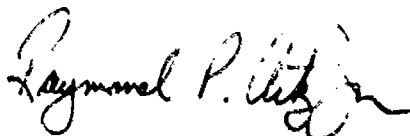
RADC-TR-88-159, Vol II (of two) has been reviewed and is approved for publication.

APPROVED: *[signature]*

THOMAS F. LAWRENCE
Project Engineer

APPROVED: *[signature]*

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

FOR THE COMMANDER: *[signature]*

JOHN A. RITZ
Directorate of Plans & Programs

RESEARCH IN DISTRIBUTED PERSONAL COMPUTER-
BASED INFORMATION SYSTEMS Semi-Annual
Technical Report No. 5   Vol II (of two)

Harry C. Forsdick
Robert H. Thomas

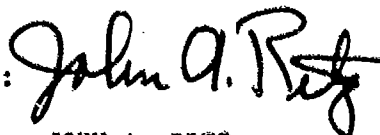| REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A

2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	

4. PERFORMING ORGANIZATION REPORT NUMBER(S) 5723	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-159, Vol II (of two)

6a. NAME OF PERFORMING ORGANIZATION BBN Laboratories Inc.	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 02238-0001	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700

8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Projects Agency	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0256

8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington VA 22304	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO. 62702E	PROJECT NO. D224	TASK NO. 01	WORK UNIT ACCESSION NO. 01

11. TITLE (Include Security Classification)

RESEARCH IN DISTRIBUTED PERSONAL COMPUTER-BASED INFORMATION SYSTEMS Semi-Annual Technical Report No. 5

12. PERSONAL AUTHOR(S)
Harry C. Forsdick, Robert H. Thomas

13a. TYPE OF REPORT Final	13b. TIME COVERED FROM Oct 83 TO Mar 84	14. DATE OF REPORT (Year, Month, Day) August 1988	15. PAGE COUNT 80

16. SUPPLEMENTARY NOTATION
N/A

17	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Multi Media Message System Distributed System Distributed Personal Computer Environment
12	07		

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

The primary focus of the personal computer task area is the development of an electronic message system called Diamond, which will run in a distributed personal computer environment. The message system will: support a user interface that exploits the capabilities of advanced single-user computers, handle messages that contain data other than text, have a distributed architecture, operate in a secure fashion, permit use from a variety of user access points, and have a transportable implementation.

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

DD Form 1473, JUN 86   Previous editions are obsolete   SECURITY CLASSIFICATION OF THIS PAGE

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This is the fifth semi-annual technical report for Contract No. F30602-81-C-0256, entitled "Research in Distributed Personal Computer Based Information Systems." It covers work done between October 1983 and March 1984. The first four semi-annual reports are BBN Reports 4924, 5301, 5395 and 5722.

1.1 Project Overview

The tasks for this project fall into three broad areas:

1. Research in distributed personal computer systems,

2. Support for the Strategic C3 Experiment,

3. Maintenance of the Hermes electronic message system.

The project objectives in each of these areas are discussed briefly below.

1.1.1 Distributed Personal Computer Systems

The primary focus of the personal computer task area is the development of an electronic message system, called Diamond, which will run in a distributed personal computer environment. The message system will.

o Support a user interface that exploits the capabilities of advanced single-user computers

o Handle messages that contain data other than text (e.g., images, line drawings, speech)

o Have a distributed architecture

o Operate in a secure fashion

o Permit use from a variety of types of user access points

o Have a transportable implementation

The personal computers used in the initial implementation of the the Diamond message system will be Jericho computer systems. Portability will be demonstrated by moving the system to another comparable personal computer system.

Development of the message system requires work in a number of supporting areas, including

1 Basic System Support. Diamond will be developed as an application program that executes on a collection of personal computers and shared resource

computers interconnected by a high bandwidth local network. Diamond, as well as other applications, requires the support of "operating system" level software. The purpose of this software is to make the Jericho personal computer usable as a sophisticated, autonomous, single-user computer system. Development of the basic system support involves the design and implementation of storage management functions, bit map display functions, a multiple process capability, an interprocess communication facility, and support for the standard DoD network communication protocols.

2. **Input/Output Support for a Variety of Data Types.** The Diamond message system will be designed to handle messages composed of a number of types of information, including text, facsimile, graphics, and speech. This capability for multiple media communication will require the development of software that supports the input and output of these different types of data, and, in some cases (speech, sound and facsimile), system engineering to interface the personal computer systems with hardware required for the input/output of this data.

3. **Distributed System Support:** Diamond will execute on a distributed system architecture Diamond and other applications developed for this environment will require supporting software designed to enable personal computers to function effectively in a multiple-computer network environment. This will include the development of a network interprocess communication facility, a distributed file system supported by personal computer storage resources and dedicated file server computer resources, means for accessing devices that are remote from a personal computer as if they were local, a user authentication mechanism, and access control mechanisms to provide for controlled sharing in a distributed environment. The software developed here will run in part on the personal computers and in part on the shared-resource computers.

4. **Programming Language Support.** The Pascal programming language will be used for much of the initial programming required for the Diamond message system development. In addition, we expect to use InterLisp for some of the user interface experimentation and research. Therefore, a certain amount of effort will be required to ensure that the implementations of Pascal and InterLisp for the Jericho computer and their supporting environments are adequate Furthermore, it is important that software modules written in Pascal and InterLisp be able to be used together in personal computer based systems such as Diamond. Currently this sort of interoperability is not possible, and it is not clear to what extent it can be achieved.

5. **Programming Environments** Diamond will be a reasonably large system. It will be built by a team of implementers, of which each member will use a personal computer for software development. To facilitate implementation of systems that will be built like Diamond, we will design, implement, and experiment with an application development environment, called the Jade environment, that is intended to support the construction of distributed application programs and that is capable of supporting programming projects large enough to require many programmers, each supported by a personal computer

We will use a new programming environment, called Aleph, to explore extensions to the Interlisp environment that exploit features unique to personal computers of the Jericho class. This will involve experimental investigation in the areas of graphical debugging, facilitation of routine

bookkeeping activities, techniques for presenting multiple views of systems, vocal annotation of textual documents, and content-addressed documentation.

1.1.2 Support for Strategic C3 Experiment

The objective of work in this area is to support the Strategic C3 Experiment, a technology transfer and evaluation project being conducted by DARPA and the Strategic Air Command. A number of contractors are working on this experiment with ARPA and SAC Our role, at present, is principally to adapt the Hermes electronic message system to the needs of SAC users who are participating in the experiment.

In particular, we are working to.

1. Modify Hermes so that it can operate with a full-screen editor, such as EMACS or WE, in order to provide full-screen editing and composing of draft messages as an integrated Hermes function.

2. Extend the data management capabilities of Hermes to provide a template-driven report generator capable of summarizing the information contained in groups of message/records.

3. Investigate the problem of software aids for scheduling personnel and equipment. Develop algorithms and experimental software to support these scheduling tasks and experimentally study user interface and implementation issues.

The work in this task area was completed during the previous reporting period. Consequently, this task area is not discussed in this report.

1.1.3 Hermes Maintenance

The objective of this task is to provide software maintenance for the Hermes electronic message system This includes correcting problems that would prevent effective use of Hermes, should any arise. installing Hermes on new hosts at the direction of the ARPA office, and making improvements to the Hermes software.

1.2 Summary of Recent Project Activity

Our accomplishments during this reporting period include the following

o First Release of Diamond. The Document Manager, Authentication manager and User Access Point software have been integrated into the first release of the Diamond system.

o Document Manager The implementation of the Document Manager progressed with the introduction of folders. access control and the ability to send documents from one user to another. In addition, we implemented the ability for multiple documents managers to share the load of storing documents and built a scavenger tool to reconstruct broken Document Manager databases.

o **Authentication Manager** The implementation of the Authentication Manager has continued Operations necessary to support the first release of Diamond have been implemented.

o **Access Point.** We have completed the initial implementations of the four parts of the Diamond User Access Point. These include the Coordinator (initial point of contact with Diamond), ShowFolder (for manipulating folders), EditDoc (for viewing and editing documents), and ShowRegistry (for managing information about users and groups of users)

o **Document Editor** We have completed the design and initial implementation of a new multimedia document editor called EditDoc. With this new editor, Diamond users will be able to view and compose documents that contain text, graphics, scanned images, voice and spread-sheet charts integrated together so that they appear to be part of one composite object. We hope that EditDoc will permit users to compose documents that have the same expressive power as journal articles or books where text, figures and captions are integrated into one cohesive document.

o **Workstation Portability Target.** Having selected the Sun Workstation as the target for porting Diamond, we have designed a strategy to move unmodified programs from the Jericho development environment to the Sun Workstation Environment This involves the development of a PCode to M68000 assembly code translator (including a peep-hole optimizer) as well as some low level routines for run-time support We have decided to port the Diamond document editor, EditDoc, as a test of this strategy.

o **Network-wide Interprocess Communication.** Work continued on the Network-wide IPC facility in the area of timeouts of interactions between Diamond components and the ability to run an operational configuration in parallel with one or more debugging configurations.

o **Software State Database** During this reporting period the initial implementation of the Software State Database was completed, and the Diamond group began using the system for all their software distribution needs

o **IPC Monitoring** We have completed the initial design of the IPC monitoring facility and have begun work on the implementation of its low-level mechanisms

o **Interlisp** Work continued on developing the Jericho Interlisp system in the areas of garbage collection, non-local exiting facilities and multiple process capabilities

o **Aleph** Work continued in the two major thrusts of Aleph Content Addressed Documentation and Programming Tools

    These items, and others, are described in more detail in the following sections of this report

1.3 Organization of this Report

The rest of this report describes our work in the task areas identified in Section
1.1 in more detail. Section 2 discusses work on the Diamond multimedia system and
related activities. Work on Jade, the Jericho Pascal operating system, is described in
Section 3. Section 4 presents our activities related to the development of the Jade
programming environment. Work on the Interlisp system for the Jericho computer is
described in Section 5. Section 6 discusses our work on Aleph. Section 7 describes
recent Hermes maintenance activity.

2. THE DIAMOND MULTIMEDIA MESSAGE SYSTEM

Diamond is a distributed system implemented by a variety of components which together provide a single coherent service. The components of Diamond are:

o **User Access Point**: The user's main contact with Diamond. The Access Point is composed of several tools including:

  * **Coordinator**: All of the actions of the Access Point are directed by this tool. The user can always inquire about the state of Diamond by interacting with the Coordinator.

  * **Document Presenter/Editor**: Documents are viewed and composed using this tool. The Document Editor embodies all of the protocols concerning Document and Atomic Object Representations.

  * **Folder Presenter**: Folders of documents and other folders are viewed and manipulated by the Folder Presenter. This tool also interacts with the Document Presenter/Editor tool to show or compose documents.

  * **User/Group Registry Presenter**: The User and Group Databases (see below) can be examined and modified using this tool.

o **Authentication Manager**: This component maintains information about authenticated users and processes of a Diamond cluster as well as long term information about user preferences and groups of users.

o **Document Manager**: Documents and folders of documents and other folders are managed by this component. When a user saves a document, the Document Manager accepts the document and stores it in a Folder for later retrieval.

o **Device Managers**: Various devices such as Image Scanners and Printers are managed by Device Managers.

o **Import/Export Manager**: Documents sent to recipients outside a Diamond cluster are Exported by this component. Likewise, documents originating outside a cluster which are addressed to a recipient supported by the cluster are imported by the Import/Export Manager. This component takes care of any protocol conversions that may have to occur between the standard DARPA Internet Multimedia Protocol and the protocols used internally by Diamond.

o **Internet Gateway**: Communication with hosts on the DARPA Internet is done by use of the Internet Gateway.

Figure 1 illustrates the architecture of a Diamond cluster.

Preceding Page Blank

Diamond Distributed Architecture
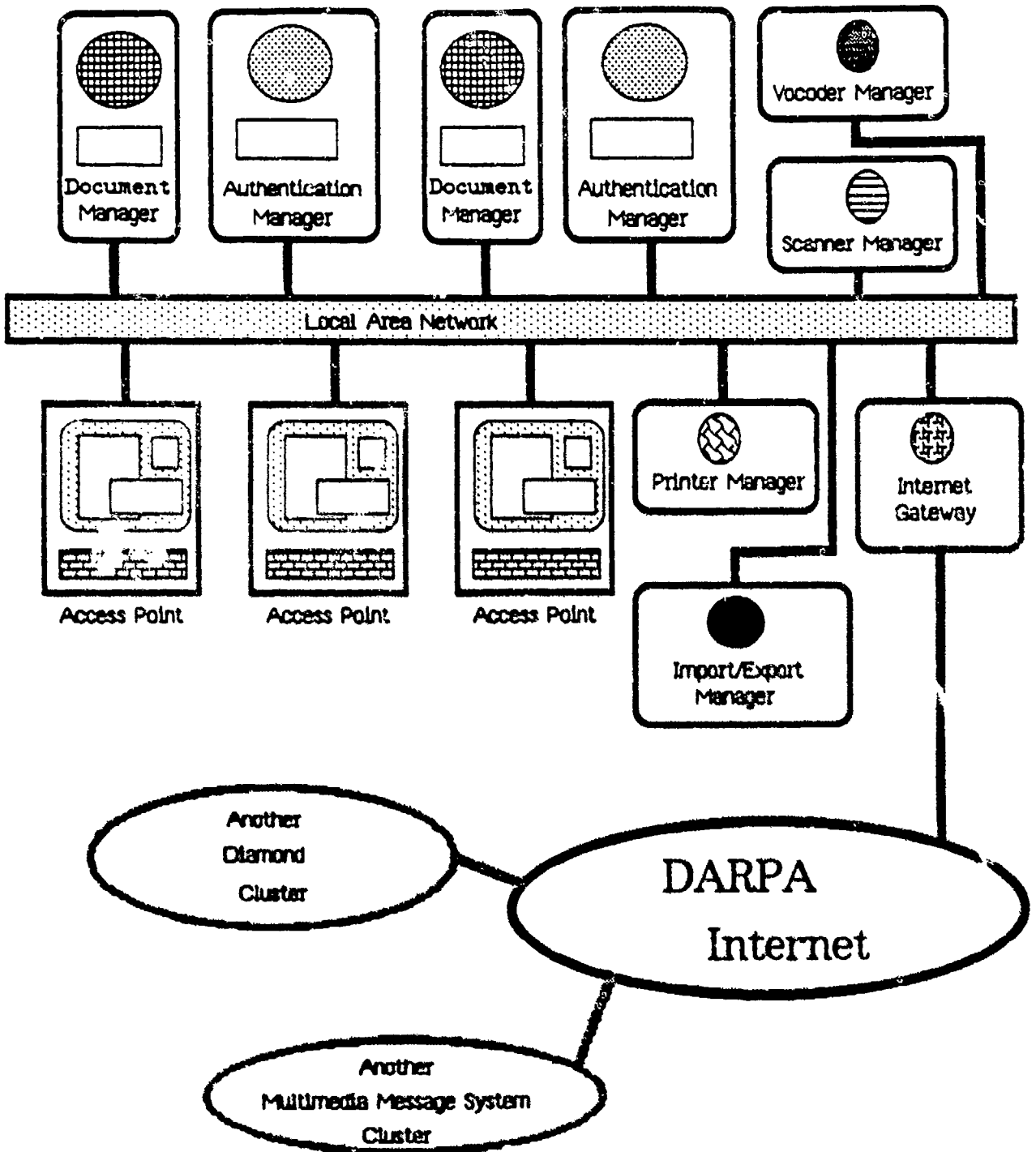
Figure 1   The Distributed Architecture of Diamond

2.1 Initial Release of Diamond

During this reporting period we completed an initial version of Diamond that runs on the distributed architecture described above. Initial implementations of User Access Point software, including a multimedia editor and tools for displaying folders and the user database, of the Document Manager, and of the Authentication Manager have been completed. With this version, user documents are stored in the Diamond Document Store, and are retrieved for presentation by the user's workstation access point. The access control mechanism designed to assure the privacy of user documents has been implemented. It is an access control list mechanism that requires cooperation between the Authentication Manager and Document Managers. While many improvements are required to make Diamond a robust and easy to use system, completion of the initial version is a significant milestone.

Since it became operational, the project staff has been using Diamond on a daily basis. As a result, a number of bugs have been identified and corrected, and a number of improvements, primarily to the Access Point tools and multimedia editor have been made.

Subsequent sections of this report describe the status of the various Diamond components shown in Figure 1.

2.2 Authentication Manager

The initial implementation of the Authentication Manager has been completed. All of the main operations on the three databases (Access Control, Principal and Group Databases) supported by the Authentication Manager have been implemented. The Authentication Manager is now used by the Document Manager and User Access Point tools in the released version of Diamond.

The Access Point needs to associate profile information with each user. For example, one user preference is the default font to be used by the various components of Diamond. A logical place to keep this information is with the user's entry in the Principal database. To do this, we have implemented *user parameters* for principals. Since similar information may need to be associated with groups, user parameters were also added to the Group database. User parameters are Name–Value pairs associated with a record in a database. Since we cannot anticipate all of the possible pairs that will be used, the representation of the pairs is extensible and expandable. The OP data representation (see section 4.2 of BBN Report No. 5722) fulfills these requirements and is used to represent user parameters. Thus user parameters are Name–Value pairs stored in the principal's or group's record in the OP data representation.

2.3 Document Manager

2.3.1 New Features

During this period, we continued the implementation of the Diamond Document Manager described in the previous semi–annual report. Several important additions were made to the Document Manager.

A-9

1. Folders were added to the document store. Folders contain references to other folders or documents; these references are called citations. Additions were made to the Document Store Manager to handle requests to add, delete, or modify citations in folders.

2. Access control was implemented. Every request to perform an operation on an object in the document store is access controlled. Examples of these operations are: creating a folder or document, reading a folder or document, sending a document to other users, and adding, deleting or modifying citations in folders. Associated with each object in the Diamond document store, there is a list of Diamond principals and the operations that they are allowed to perform on that object. When a request to perform an operation is received by the Document Manager, the Document Manager contacts the Authentication Manager to obtain the Diamond principal associated with the process making the request. The Document Manager then checks that principal's access against the list of principals and access rights associated with the object and performs the operation only if that principal is allowed to perform the requested operation.

3. The ability to send documents as messages was added to the Document Manager. When an user is created in the Diamond system, two folders are created for the user, a root folder which contains all the other folders belonging to the user and an InBox folder, which is used to receive messages. A document is sent to other Diamond users as follows. The Document Manager parses To and Cc lists associated with the document in o der to obtain the names of the recipients. It then contacts the Authentication Manager to obtain the UID of the InBox folder associated with each of the recipients. Finally, it places a citation for the document in each InBox folder. Note that the documents are never copied. The Document Manager can currently send messages to other Diamond users; it will be expanded to send messages to users outside the Diamond system and to send messages to groups of Diamond users.

2.3.2 Multiple Document Managers

During this period, we also began work on multiple Document Managers. A Diamond cluster will contain one or more Document Managers, each running on a different machine and managing a document store on that machine. Multiple Document Managers will eventually be used both for load balancing, i.e. a folder or document will be stored on one of several document stores to limit the load on any one machine, and for reliability, i.e. key folders and documents will be stored in more than one document store to make them more readily available in the event of machine crashes.

The Diamond document store is organized as follows. There is a root folder which is the base of the Diamond hierarchy of folders. The root folder contains a citation for the root folder of each Diamond user, the root folder of each Diamond user contains a citation for that user's InBox folder and for any additional folders that the user chooses to create in their root folder. The user can create folders in their root folder or in any folder in their root folder, thus, the user has complete control of their own hierarchy of folders.

There are currently two Document Managers running in our Diamond cluster. These currently implement load balancing, but not replication of folders and

documents. The load balancing is implemented as follows. One document store contains the Diamond root folder. When a new Diamond user is created. the Authentication Manager, which creates new users, contacts each Document Manager to obtain the size of that manager's document store. The Authentication Manager then creates the user's root folder in the smallest document store; any folders subsequently created in that user's root folder are created on the same document store. In other words, load balancing is achieved at the Diamond user level; some Diamond users have all of their folders on one of the document stores and some have all of their folders on the other document store. Note that where their folders are located is invisible to the user.

Implementation of multiple Document Managers required several enhancements to the Document Managers. For example, when documents are sent as messages between users whose folders are on different document stores, it is necessary to add a citation for a document on one document store to a folder on another document store. In order to support multiple Document Managers, the Document Managers were modified to generate requests to and process requests from other Document Managers. Examples of these requests are adding and deleting citations from folders and modifying reference counts of objects stored on one document store and referenced on another document store.

During this period, we also studied the problem of load balancing on a more detailed level. i.e splitting one user's folders across multiple document stores, and the problem of replicating folders and documents. We plan to add both these features to the Document Manager.

2.3.3 Document Cache

In order to improve performance of the Diamond system, we implemented a local cache for the document store The cache is maintained at each access point and contains the folders, documents. and atomic objects most recently referenced from that access point. Whenever an user requests that a folder, document or atomic object be retrieved. the access point software checks the contents of the local cache. If a document or an atomic object is requested and the cache contains it, the document or atomic object is simply read from the cache. Documents and atomic objects are immutable, thus a copy of the object in the local cache is guaranteed to be the same as the copy in the document store. If a folder is requested, then the access point software contacts the document store to determine if the cached copy is up-to-date. If it is up-to-date. the cached copy is used. if it is not, the document store returns the up-to-date copy of the folder and it is stored in the cache. A tool was implemented to control the size of the cache (i.e. the number of disk pages used by the cache) by deleting the least recently referenced objects from the cache. This tool is typically run periodically by a background process.

2.3.4 Document Store Scavenger

We implemented a scavenger for the document store. The scavenger is used to check for document store inconsistencies caused by software bugs or hardware crashes and to correct them. The scavenger reads every folder and document in the document store and constructs a table of the object's UID, objects referenced by the object, and objects that reference the object. This table is compared to the document store database. and the database entries are corrected to match the table. The scavenger can handle multiple document stores. Currently, the Document Store

managers must be stopped in order to run the Scavenger. We plan to improve the scavenger so that the document store can be dynamically scavenged while the Document Store managers are in use.

2.4 Access Point

As reported in the previous semi−annual technical report, users access Diamond through Access Points. Diamond is designed to accommodate access points with a range of capabilities, from powerful personal computers connected to high performance local area networks ("high end" access points) to alphanumeric terminals with modems ("low end" access points). In this contract, we have been focussing on the "high end" access point design and implementation. During this reporting period, we have implemented the various parts of the Access Point for the Jericho. The implementation has closely followed the design document that was described in the last report. The Access Point software will be ported to Sun Workstations later.

There are five principal parts of the Access Point visible to the user:

1. **Session management functions**: Login, Logout, establishing user environment from profile, status, and cleanup.

2. **Folder management functions**: Viewing folders, citation manipulation, and access control to folders and documents.

3. **Document management functions**: Viewing and editing documents.

4. **Principal and Group management functions**: Viewing, editing, and access control of Principals and Groups.

5. **User Profile management functions**: Viewing and editing user profiles.

To achieve the goals of the Diamond user interface, we have adopted a multi−process architecture that allows multiple concurrent tasks to occur. The Access Point is made up of a collection of tools (Folder Presenter, Document Presenter, Principal/Group Presenter) and a Coordinator program which provides session management functions and coordinates the activities of the various access point tools. Each tool runs in a separate process and window to allow maximum user flexibility.

The Coordinator program is the controlling element of the user interface; it principal responsibility is managing the display and the various windows used to present and edit documents and other objects. Since users gain access to the system by invoking the coordinator, the coordinator is called Diamond. The coordinator also provides login control (including re−authentication in the event of a failure) and profile management. The user can tailor Diamond to suit his/her own needs. Using profile preferences, a user can control the area of the display surface to which Diamond will confine its operations, the size and shape of windows used to display folders and documents, the format used to display citations in folders, the fonts used in various situations, and a variety of other aspects of Diamond operations. A user's preferences are stored in the user's (principal) record in the User Registry, and they are obtained from the Authentication Manager when the user logs in.

The library APLib was developed to support common functions in the access point tools. APLib interacts with the Coordinator to perform most of the functions it provides. Among the functions is a set of global commands (a menu hierarchy) which is available from any access point tool.

The folder presenter tool (called ShowFolder) provides the user with the ability to display and manipulate the contents of folders. A number of features are under user control through preference settings, including: the display format of citations, the set of citations displayed, and the order of citations displayed. ShowFolder also provides the user with an access control list editor.

The document presenter tool (called EditDoc) is described in the next section.

The principal/group registry presenter tool (called ShowRegistry) provides the user with the ability to display and manipulate the principals (i.e., the list of groups a principal belongs to), groups (i.e., the list of principals in a group or the list of groups a group belongs to), the principal registry (i.e., the list of principals), or the group registry (i.e., the list of groups). ShowRegistry also provides the user interface for creating new principals and groups.

The initial version of the complete access point became available in February, 1984, and has received steady use since. We are continuing to improve its performance, its functionality, and its ease of use.

2.5 Multimedia Document Editor: EditDoc

We have completed the design of a new multimedia document editor for Diamond called *EditDoc*. Earlier in the project we developed a multimedia editor, known as MMEdit, which allowed us to experiment and test ideas concerning multimedia editors. One of the main conclusions to come out of the MMEdit experiments is that a multimedia editor should support documents in which all of the various multimedia elements are laid out and directly visible on one display surface. Such an editor would simulate a single piece of paper (or perhaps a sequence of pages) on which different multimedia elements have been placed. To do this properly, each of the sub-editors for the different media types must observe a common set of conventions and implement a set of generic operations.

EditDoc has been designed based on the conclusions from the experiments with MMEdit. With EditDoc, Diamond users will be able to view and compose documents that contain text, graphics, scanned images, voice and spread-sheet/charts integrated so that they appear to be part of one composite entity. We hope that EditDoc will permit users to compose documents that have the same expressive power as journal articles or books where text, figures and captions are integrated into one cohesive document.

In EditDoc, a document is a collection of elements (of possibly different media type) each of which is located at a particular position on a quarter plane (extending infinitely to the right and down) and occupying a specified width and height. In most instances, users limit themselves to producing documents which have conventional widths. The window in which EditDoc displays the document is a viewport on the quarter plane and may be positioned, using scroll bars, anywhere on the quarter

plane. Currently, we do not allow elements to overlap, although there is no reason why they could not. We have not addressed the issue of pagination of documents in the design of EditDoc, although this does not appear to be a difficult problem and will be dealt with in the future.

One of the goals of MMEdit was to be as extensible as possible, if a new application program was developed, it was possible to include the display of that program as part of a document. With such "general purpose" elements, the presence of the element was indicated with a text caption and the object could be viewed by asking for more detail about the caption. More detail would be provided by creating a separate window and then running a separate application program on a data file that was included in the message. In this way, spread sheets were included in MMEdit documents by invoking the spread sheet program. Any other useful display produced by an application program could be included in a document. Although this approach to integration is simple and all-inclusive, it is lacking in the area of document integration. People viewing such documents were distracted by the fact that general purpose elements were displayed differently from text, images and voice elements which were supported directly by the editor.

As a result, in EditDoc we relaxed the requirement that any display produced by any application be able to be included in a document. Instead, we have defined a specification that must be met by programs that implement multimedia element types to be included in EditDoc documents. Thus, EditDoc is a multimedia document editor which calls on sub-editors to manipulate mono-media elements of the document. The main editor, EditDoc, interprets commands whenever the mouse arrow in in the "white space" of the document while the component editors interpret commands when the arrow is in a box corresponding to the media type that they support.

New element types can be added to the multimedia document editor either by writing a new sub-editor from scratch following the sub-editor specification or by creating a front-end to an existing application. The front-end provides the functionality required by the sub-editor specifications and uses existing functions of the application to achieve the desired effect. We will take both approaches in the implementation of EditDoc. Specifically, we already have an implementation of a line-drawing graphics program to which we will add a front-end to provide the EditDoc functionality.

The EditDoc sub-editor specification is as follows:

1. The sub-editor is completely responsible for the creation, presentation and editing of elements of a given type in a multimedia document.

2. The display of the element either for presentation or editing should be confined to the dimensions of the box allocated for the element by EditDoc.

3. For each element in a document, EditDoc will maintain a record which describes the type of the element, the position and dimensions of the box surrounding the element and a field which points to a sub-editor managed data structure which is the type-specific representation of the element.

4. The sub-editor should support the following generic functions which will be called with an element of the supported type as an argument:

Init*ElemType*   Initialize any global data structures of the sub-editor.

This is the only function which does not take a specific element as an argument. It is called once per invocation of EditDoc.

Enter*ElemType* — Create a new element of type *ElemType*. Do any initialization of the sub-editor specific data structure associated with the element.

Edit*ElemType* — Edit an existing element. While editing, a set of conventions must be followed so that the user perceives an integrated interface to EditDoc. These conventions are described below.

Draw*ElemType* — Draw the display of an existing element.

Delete*ElemType* — Delete an element from the document. Release any storage occupied by the sub-editor specific data structure representing the element.

Write*ElemType*BoxToOctetBuf — Save a representation of the element in an array of octets. This is the way individual elements of a document are saved on non-volatile storage.

Read*ElemType*BoxFromOctetBuf — Take an uninitialized element and initialize it with the contents of an array of octets that was produced by a previous call to Write*ElemType*BoxToOctetBuf.

CB*OtherElemType*To*ElemType* — This describes a family of functions that are used by the Diamond Clipboard mechanism to translate from one element type to another. If there are N types of elements in an implementation of EditDoc, then there will be $N*(N-1)$ such functions, although some of them may be empty if it is not possible to perform the translation.

CB*ElemType*Reassign — Reassign the storage pool used by the sub-editor data structure used by an element on the Clipboard.

CB*ElemType*Dispose — Dispose of the storage occupied by the sub-editor data structure used by an element on the Clipboard.

5. The set of conventions to be followed while editing an element are as follows:

   a. If a button is clicked on the mouse while editing, then examine where the mouse is located. There are three cases:

      i <u>Mouse is in box surrounding an element</u>: Depending on the button pushed, perform the appropriate action. If the left button is pushed, then select a part of the element, whatever that may mean for the particular element type. If the left button is held, the mouse dragged and then the left button released, then the

select operation is performed on multiple parts of the element. If the middle button is pushed, then pop up a menu and allow the user to select an operation. If the right button is pushed, then, depending on the context either abort the currently uncompleted operation or, if there is no operation outstanding, unselect all selected parts of the element.

    ii. <u>Mouse</u> is <u>in</u> <u>scroll</u> <u>bar</u>: Call the routine DoScrollBarCmd. This will cause the viewport on the document to be moved to some other position.

    iii. <u>Mouse</u> is <u>somewhere</u> <u>else</u>: Return to the main editor EditDoc after saving any state that must be preserved.

6. If a key is typed on the keyboard then interprete that key as a command to the sub-editor. For some sub-editors, keyboard input is not appropriate and keystrokes will be ignored. A graphics editor is an example. it is difficult to do graphical manipulations from the keyboard; the mouse pointer is much better. Other sub-editors, such as the text sub-editor, will rely heavily on keyboard input, since that is the primary way to enter text. Still others, such as the spread-sheet/chart sub-editor adopt a combined strategy using the keyboard where it is the best input device and the mouse pointer where it is best.

7. If any other message comes to a sub-editor, it should suspend its operation and return to the main editor. This is different from leaving the sub-editor since the appearance of the display should not change and the sub-editor will be returned to by the main editor after the message has been processed

There is another level of detail to the specification, however this specification imparts most of the spirit of how the main editor EditDoc and the sub-editors will interact.

The initial implementation of EditDoc has been completed and currently there are sub-editors for text. graphics, speech and images. We have plans for adding a sub-editor for spread sheet/charts.

2.6 Import/Export Manager

We have developed an initial implementation of the Diamond Import/Export Manager. This component is responsible for supporting message communication with users who are external to a Diamond cluster. It takes messages addressed to external users and translates them into either SMTP (text-only) format or RFC 759/767 (multimedia) format, depending on information in the User Registry (external multimedia recipients have an MPM address stored in the registry). It then submits the translated messages to a delivery service that transmits them to the appropriate internet host(s). In addition, it accepts messages which originate outside of a Diamond cluster, translates them to Diamond format, and delivers them to users local to the cluster.

Since the Import/Export Manager handles both multimedia and text-only mail, it

can translate (reduce) a multimedia message originated within Diamond to a text—only message suitable for delivery to users at text—only sites. It does this by retaining the text parts of the original message and replacing the non—text parts with indicators that record the type of object that was removed. This permits us to use Diamond for all of our daily message activity, both text—only and multimedia.

2.7 Printer Manager

The Printer Manager is used to produce hard copies of Diamond documents. Because of the variety of media types that can appear in Diamond documents, we require printers that can reproduce arbitrary bitmaps. Laser printers are the most attractive alternatives because of their speed and the quality of their output. The work of this manager involves a translation between the internal representation of documents used in Diamond and the protocol for driving the laser printer.

We have performed some initial experiments with producing hardcopies of Diamond documents on a laser printer, the QMS model LG1200 laser printer. A number of issues were raised by this experiment, including:

o The set of fonts available on the laser printer is different from the set used in Diamond. While it is possible to make conservative mappings between the two sets[1], this is not a satisfactory solution because the appearance of the printed document is different from its appearance on the display. Another approach would be to down—load the Diamond fonts to the laser printer and use them in the printing process. The problem with this approach is that the resolution of the Display device on which the Diamond fonts are displayed is one third the resolution of the laster printer. Thus Diamond fonts appear to be very small on the laser printer. Automatic scaling programs do not work very well either, because most good digital fonts have had the attention of a good type face designer to eliminate staircasing effects of raster devices.

o Similarly, the line drawing graphics capabilities of the QMS LG1200 (and most likely all other laser printers) are different from the representations used in Diamond. For example, in Diamond, it is possible to specify arbitrary fill patterns while on the QMS LG1200 there are a set of fixed fill patterns. In any case, the difference in resolutions would also impact the appearance of textures.

Future work on the Printer Manager will have to address these issues. Since we are not certain about what laser printer will be used in the future with Diamond, we are currently deferring work on the Printer Manager.

---

[1]conservative in the sense that letters in the font on the laser printer occupy the less than or some space as letters in the corresponding font in Diamond

A-17

2.8 Porting Diamond to the Sun Workstation

During this reporting period we completed our decision on the strategy to be used to port Diamond to the Sun WorkStation. The approach adopted for porting Diamond is to translate the Jericho Pascal compiler's *PCode* output into M68000 assembly code. By using this approach, the vast majority of Diamond and its support libraries can be ported *as is*. Most of the machine differences are accommodated in a small number of libraries which have identical interfaces for the two machines but differ in their implementations.

The initial version of the translator to perform the conversion from PCode to M68000 assembly code has been completed. This version is lacking certain code generator optimizations but is otherwise fully functional.

The successful operation of the translator required a few minor modifications to the Pascal compiler. One such modification was to provide for the compilation of IEEE floating point constants which are used on the Sun instead of Jericho floating point constants. A compile-time switch determines which conversion is performed.

The runtime support for such functions as access to files, free storage facilities, and inter-process communication have been written. Several libraries have been translated including most of those which require different implementations on the Sun.

As a test vehicle, we have ported EditDoc which is the document editor component of Diamond. EditDoc was chosen because it is a program of substantial size which makes heavy use of the display primitives, uses files, and uses the mouse, but does not need to use the network communication facilities. Porting EditDoc required that several standard libraries be ported and that has been done. Most of these libraries were simply recompiled and translated and required no modifications.

The conversion of some libraries posed particular problems and these are discussed below.

2.8.1 Interprocess Communication

One of the major differences between Jericho and the Sun Workstation is that Jericho is a single address space, multiple process machine and the Sun Workstation is a one address space per process machine. The Jericho interprocess communication (JIPC) mechanism depends heavily on memory sharing as a mechanism for passing information from one process to another. The Sun Workstation running Berkeley 4.2 UNIX provides interprocess communication through character streams between processes. We have mimicked JIPC on the Sun workstation by passing the data contained in the JIPC messages plus the data in the storage block whose address was being passed in the JIPC message through the Sun IPC character stream and reconstituting the storage block on the receiving end.

There are several instances where actual data sharing is used by two or more processes. One of these is the extended variable data base This mechanism allows values to be associated with particular keywords as a way of providing parameters global to several programs or for avoiding repeated complicated computations from

session to session. Since, on the Sun workstation, each process has a separate address space, the values of variables set in one process are not available to any other process. To circumvent this, a server process maintains the database and responds to queries and requests from other processes via the mimic JIPC. We expect there will be additional such servers needed for similar situations.

2.8.2 Windows

Several problems related to the use of windows were encountered. The mechanism in the Sun window system for always maintaining the correct contents of a window is too costly to use. It requires doing every operation twice: once on the display screen and again on a backup copy of the window contents in main memory. This slows down painting operations by at least a factor of two. The alternative is to respond to interrupts indicating that particular areas of the window have been damaged by being overlaid with another window and repainting those areas from the underlying data.

Responding to these window damage interrupts requires substantial alterations to the control structure of EditDoc and its libraries. In particular, it must be possible to back out of those operations, such as selecting an item from a menu, which wait for a user action. This is because such library routines have no knowledge of what information is supposed to be on the screen and cannot be expected to repaint it. They must return to the main program to allow it to repaint the screen.

2.8.3 Mouse

The Sun window system does not provide any way for a program to determine the position of the mouse. The only way to know the position of the mouse is to receive a continuous stream of input events each reporting the mouse position. This might suffice if only one process needs to know where the mouse is, but, in fact, several processes may need to know. Our solution is to do the mouse tracking in the window manager process and for other processes to query the window manager via JIPC to determine the mouse position. Such queries are time consuming (roughly 30 msec apiece versus a few microseconds on Jericho) and will severely impact certain operations.

Another problem with the mouse is the limited size of the cursor image (16 by 16 pixels). We have designed new icon fonts to accommodate this, but a larger image would be preferable.

The method of displaying the mouse image on the screen is also deficient. It is not possible to insure that the mouse will always be visible regardless of its background. On Jericho, the mouse image can consist of both a white part and a black part. This allows the white part to be visible on black backgrounds and the black part to visible on white backgrounds. The Sun mouse image is restricted to a single image which can be white or black or can complement the background. None of these can be guaranteed to be sufficiently visible over all backgrounds. No satisfactory solution to this problem has been found.

2.9 Papers and Presentations

We have produced a video tape which presents the goals, architecture, and equipment used in Diamond. It illustrates the user interface to Diamond by showing how a user reads a message and then how he would compose a response to that message.

We have submitted a paper, titled "Initial Experience with Multimedia Documents in Diamond" to the IFIP WG 6.5 Working Conference on Computer-Based Message Services to be held in May in Nottingham, England. The paper describes and compares three different approaches to dealing with multimedia documents: the approach used in the experimental multimedia document editor MMEdit, the approach used in the DARPA Internet multimedia protocol and the approach being used in the EditDoc multimedia editor.

3. THE JERICHO JADE SYSTEM

Jade is the operating system for the single user Jericho computer system developed at BBN. This section documents tasks that were performed in improving the Jade system primarily to meet the requirements of Diamond.

3.1 Performance Improvements

Several enhancements to the RoutineTrace tool and Pascal microcode were made to help us track down performance problems. Briefly, RoutineTrace is a tool which initiates and terminates the recording of certain information about what programs are doing and then analyzes and presents that information to aid in determining where bottlenecks and other performance reduction is occurring. The enhancements were to add additional entries in the data being recorded about page fault behavior and present that information in the RoutineTrace printout.

The information derived from these printouts lead to revisions to the scheduler algorithm. What we discovered was that processes could consume their entire run time quantum by faulting in a few pages and then some other process would be run. This process might then exhibit the same behavior. If a sufficient number of processes were runnable, they might all consume their entire run time quanta with very little useful work being done and then by the time the first process made its way back to the head of the priority queue the pages it had faulted in would have been removed to make way for the other processes' pages. Severe thrashing would ensue. The modification that was made was to credit a process's run quantum for the time lost to page faults. Thus the run quantum would determine useful time spent rather than total time spent.

3.2 Synchronization

A synchronization mechanism has been added to Jade which implements the notion of mutual exclusion semaphores and the P (lock) and V (unlock) operations. The synchronization mechanism was motivated by the needs of Diamond. In particular, managers such as the Diamond Document Store and the Authentication Manager have multiprocess implementations which enable them to service multiple simultaneous client requests. A general purpose synchronization mechanism was required so that processes within a manager can acquire exclusive access to various internal databases.

Two possible implementations of the synchronization mechanism were considered. The first uses a server process and the low level interprocess communication mechanism. The second uses shared memory and the interprocess communication mechanism. The second is more efficient, and hence is being used on the Jericho. However, the second approach will not work on the Sun Workstation because the Sun has no shared memory. The first approach will be used on the Sun.

3.3 WindowSystem

The Jade window system has been enhanced to include a "hint" facility. A hint is a signal sent to a process that an event relevant to a window of interest to that process has occurred. Examples of such events include selection of a window (e.g., the user has selected the window for keyboard and mouse button input), de-selection of a window, and adjustment of the shape of a window. When a process receives such a hint signal, it can take action appropriate to the event. For example, when the window is selected, the process might choose to highlight various regions of the window, and when the window is de-selected, it might choose to "un-highlight" them. A process dealing with windows can request hint signals for various events from the Window Manager or it can instruct the Window Manager not to provide hint signals. The hint mechanism was developed to support the needs of the Diamond Access Point, which must permit a user to switch back and forth among multiple windows.

3.4 Improvements to Pascal Debugger

The Pascal Debugger support for breakpoints has been improved in a number of ways during this reporting period. A list of all the breakpoints that have been set can be obtained. All breakpoints can be removed by a single command. Breakpoints can be named such that when a named breakpoint is encountered the debugger reports its name; in addition, breakpoints can be removed by name. When a breakpoint has been encountered, it is possible to resume execution from (goto) another point in the routine containing the breakpoint (or any routine within the dynamic scope of the routine containing the breakpoint); it is also possible to resume execution from a breakpoint by exiting from a specified routine that is on the stack beneath (i.e., that had been called by) the current routine.

4. THE JADE PROGRAMMING ENVIRONMENT

The Jade programming environment is the set of tools and libraries that is used to develop applications on a set of distributed single-user computer systems such as Jericho or Sun Workstations. This section describes our efforts during this reporting period to enhance the ability to produce distributed applications on such workstations.

4.1 Network Protocol Software and IPC

4.1.1 Internet Protocol (IP)

The Jade implementation of the DoD Internet Protocol (IP) has been changed to provide a software "loopback" for internet packets sent by processes to their own local host. Previously such packets were sent to the nearest internet gateway, which simply returned them to the local host. Sending the packets to a gateway was an implementation shortcut that permitted the host to treat the packets as if they had originated at some other host. For example, if a TCP connection were opened between two processes on the same Jade host, the data packets sent on the connection would be "reflected" off the nearest internet gateway.

With the software loopback such packets no longer leave the local host, but rather are processed as if they had come in from the network. This reduces the load imposed on gateways for local traffic, it reduces the delay for such local communication, and it also makes such local communication possible when no gateway is available.

A more general network routing mechanism has been implemented which performs three functions: special routing, message redirection, and segment size control. Special routes can be set up so that packets addressed to a particular host or network (or protocol) can be transmitted to a particular local network address. This is the mechanism whereby the loopback feature is enabled or disabled. It also permits a specific gateway to be used to get to another network rather than the default gateway.

The routing table is used to implement the message redirection feature. When gateways receive messages for which they are not on the optimum route or when hosts receive messages via a non-optimum interface, they send ICMP redirection messages to the originating host. The improved IP implementation makes use of these messages to modify the routing table so that subsequent transmissions to such destinations will be sent to the preferred local address.

Associated with each routing entry is a maximum segment size for segments being sent to that destination. This allows a segment size to be chosen which will avoid fragmentation and reassembly costs. It also permits proper fragmentation to be performed when hosts on the local network vary in the packet size they can handle.

The new IP implementation also maintains a table of hosts, networks, and

protocols which are unreachable. When ICMP destination unreachable messages are received from gateways or hosts, an entry is made in this table. Subsequent attempts to transmit to that destination will fail rather than cause another transmission. This reduces extraneous network traffic and permits protocol implementations built on IP (such as TCP) to quickly indicate failure rather than retransmitting for a period of time and then failing with an uninformative result. The entries in this table are aged such that after roughly a minute or two they are removed. This allows the recovery of a once dead host to be noticed and communication to commence.

The IPStat program was modified to provide information on the information in the routing and dead destination tables. There is currently no dynamic way of adding special routes to the routing table, but special routing entries can be put in the >Configuration file and these will be entered when the IP server starts up.

4.1.2 File Transfer Protocol (FTP)

Our last semi—annual report described the Jade implementation of the TCP—based DoD standard File Transfer Protocol (FTP). During this reporting period the TCP—based FTP for Jade has been made sufficiently reliable that we now use it for file transfers between Jerichos and non—Jericho hosts in place of the private file transfer protocol we had been using[2] . In particular, we use the TCP FTP exclusively for transferring files between Jerichos and our TOPS—20 hosts.

4.1.3 Interhost Interprocess Communication (IPC)

The IPC facility developed to support Diamond and other distributed applications has been described in previous semi—annual reports. During this reporting period, the interhost IPC facility has been enhanced in two ways: features have been added to help processes use timeouts more effectively, and a mechanism has been developed to permit multiple disjoint IPC "configurations" to run on the same network.

Support for Managing Timeouts

Diamond components use timeouts as a means for monitoring interactions with other components. For example, when a User Access Point process attempts to retrieve a multimedia document from the Document Store, it sends a request message to a Document Manager and waits for the Document Manager to respond by transmitting the requested document. It uses a timeout so that it doesn't wait forever for the response.

Although the timeout mechanism provides a means for a component to detect the possible failure of another component, it presents two problems: the selection of an appropriate timeout interval, and the action to take if the timeout occurs. The timeout interval must be small enough so that possible failures are detected quickly, but large enough so that timeouts don't occur under heavy load conditions. When a timeout does occur, a component is faced with the problem of determining whether the timeout was due to a failure, in which case it should give up or try another manager, or was due

---

[2]The private FTP is described in "Research in Distributed Personal Computer—Based Information Systems", BBN Report No. 4924.

to an unexpectedly large amount of processing necessary to handle the request or to heavy load conditions, in which cases it should continue waiting.

Message tracing and host probing capabilities designed to make it possible for a client process, such as an Access Point tool, to determine the status of another host have been implemented. These mechanisms are intended to be used together with timeouts to achieve improved reliability and failure recovery. In particular, when a timeout occurs they make it possible for a process to determine if the host to which a (request) message has been sent has crashed, has crashed and recovered since the message was sent, or is simply slow in processing the request.

These mechanisms make use of a host "incarnation number" scheme. Each host has an "incarnation number" associated with it that is incremented each time the host (actually the IPC software on the host) is restarted. When IPC components on different hosts communicate to send and receive (reliable) messages they also exchange and store each other's incarnation numbers.

The host probe mechanism provides clients means to determine whether a remote host is up, and if so, its current incarnation number. This is used by the higher level Diamond software as follows.

When a client process, such as the Access Point, invokes an operation on an object (requesting that the message be sent reliably rather than with minimal effort), the IPC makes available to it the host to which the operation was sent and that host's incarnation number. Should a timeout occur, the process can initiate a probe of the host to which its request was sent to determine what action it should take. After initiating the probe, it should restart its timer and wait for either the result of the probe or the response to the operation. The client's probe request is handled by the IPC which attempts to probe the IPC at the destination host to obtain its incarnation number. The destination IPC will respond with its incarnation number if it is up, or, if it is not, the local IPC will time out the probe and declare the destination host to be down. In either case, the local IPC will deliver the result of the probe to the local client process.

When the probe result is delivered, if the result indicates either that the remote host is down or that it is up but has a different incarnation number from the one it had when the requested operation was initiated, the client can assume that the operation did not successfully complete, and it should take an appropriate recovery action. If the remote host is up and its incarnation number is the same as when the operation was initiated, the client can assume that the operation is still being processed by the manager and that it will eventually complete. Therefore, the client process can continue waiting. Of course, should the time out occur again, the client should initiate another probe of the host. The assumption here is that the manager performing the operation will eventual complete it and reply to the client. For this scheme to work, managers must be well-behaved and care must be taken when shutting down a manager to ensure that no client requests are pending.

The IPC facility has also been modified to deliver a negative acknowledgement to a process sending a message whenever it determines that the message cannot be delivered. This enables client processes to detect failures before their timeouts occur, making them appear more responsive to users.

Multiple Configurations

It is desirable to be able to run more than one group of IPC Servers on a single collection of hosts at the same time, such that an IPC Server within a group supports communication among processes within that group and IPC Servers in different groups do not communicate. The same host might have several IPC Servers running on it, each of which is a member of a different group or configuration. For example, there are situations where it is useful to have several debugging configurations and an operational configuration sharing the same network and set of hosts.

Running multiple configurations on the same hardware .base requires means for ensuring that components within a configuration limit their activities to their own configuration. It is useful to think of the IPC and client/manager processes as operating at different levels. Different techniques for achieving isolation between configurations may be needed at different levels within a configuration. At the IPC level isolation can be achieved by preventing communication between configurations. At the client/manager level achieving isolation may require ensuring managers in different configurations don't interfere with one another by trying to control the same devices or by accessing the same databases.

At the IPC Server to IPC Server level, running multiple simultaneous configurations requires that a given IPC Server be able to distinguish IPC Servers that are in its configuration group from those that are not. The manner in which IPC Servers use UDP and TCP to communicate makes this easy to accomplish. Isolation of groups can be accomplished by assigning each group different UDP and TCP ports.

During this reporting period we modified the Jade IPC Server to implement the scheme for partitioning IPC configurations based on the use of UDP and TCP port. The ability to support multiple configurations has been proven to be very useful. It enables us to have an operational configuration of Diamond that runs all the time, and to run one or more debugging configurations, as required at various times to support system development.

4.2 Software State Database

During this reporting period the initial implementation of the Software State Database (SSD) was completed, and the Diamond group began using the system for all their software distribution needs. The following sections describe our initial experience with the system.

4.2.1 Software Distribution

The biggest improvement which we have noticed in using the system is in how quickly and quietly new software gets distributed. This process has speeded up both the release and the subsequent retrieval of new files.

o Release.   Before the Software State Database was installed, releasing new software was a side-effect of the process of updating to get the latest version of all software. Since this was a rather long process, programmers would tend to batch releases with updates which took place every few days.

Under the new system, installing new software is separated from the update process and can be done quickly and painlessly.

o Retrieval. Updating to get newly released software has become faster and more fully automated. The speedup has occurred since the files to retrieve can be determined by examining a single database file rather than examining the entire file system and comparing write dates and version numbers.

The functionality of several programs, some of which were necessarily interactive, has been combined into the simple *BringOver* operation. Most users now have a background request which runs at night and does a complete update automatically. It is also easy to use the interactive interface to do a partial update when necessary.

This improved functionality has been especially useful since we have begun work on a number of closely—coupled systems which are being concurrently developed by several programmers. It is important that they be able to quickly exchange and coordinate the release of new versions of their dependent modules.

Another major advantage of the database approach is that it is now possible to look in one place and determine which version of some module exists on each machine in our environment. The software database maintains a separate version number for each machine installed in the system. It is possible to look at the database and determine who needs to do a retrieval in order to get some newly released software. This is especially useful as distributed applications come on line, since these may require all machines to be running a consistent protocol. We have made heavy use of this capability in distributing new versions of the Software State Database itself, since there have been several instances where the protocol between the central database controller and client programs has changed or the format of records in the database was altered.

4.2.2 Concurrency Control

The Software State Database system provides a distributed CheckOut, Install, CheckIn facility. Before modifying a module, the programmer must CheckOut the module using SSD. While the module is checked out, no one else can check out or install a new version of that module. Upon completing the changes, the programmer Installs the new version and checks the module in. It is now available for other programmers to check out and modify. A programmer can easily determine the status of a file by examing the distributed database.

In the period leading up to the initial release of SSD, our group began development of several systems which required close interaction between multiple programmers. There was significant contention for several modules containing declarations used by all the sub—systems. The release of the SSD system made the process of sharing these files simpler in several ways.

1. It was easy to determine if another programmer was in the process of modifying a file. Before the release of SSD this would require a flurry of phone calls or walking around to several offices.

2. It was easier and faster to make a new version available. The Install procedure informs the database about the existence of a new version and

moves a copy of the newly installed file to a central repository. At that point, anyone interrogating the database would get the latest version.

3. When checking out a file, SSD ensures that the latest version of the file is on the file system servicing the user checking out the file. Previously, a user would do a full, time-consuming update in order to ensure the latest version of everything was on his machine before working on a single file.

4.2.3 Distributed Architecture

The Software State Database was the first major system to make extensive use of the full Inter-Process Communication system and Operation Protocol developed to support Diamond's distributed architecture. Our experience with SSD resulted in a number of changes to the IPC mechanism, particularly in the area of robustness in the face of failures.

One database controller resides on a single machine within a cluster of machines connected by a local area network. This controller maintains a master copy of the Software Database and serializes all requests which modify the database (operations such as Install or CheckOut). A request to perform some operation is formulated on the user's machine and then sent off to the database controller using the IPC mechanism. This request is then confirmed or aborted by the controller, depending on the state of its master database. The appropriate response is sent off to the requesting machine and, upon confirmation, any file movements associated with the request are done by the user's machine. Upon completion of the file movements, the user's machine again contacts the database controller to notify it of completion of the request. At this point the changes to the database are made visible to other users of the Software State system.

Though the view of the database which exists on any user's machine is just a snapshot of the true status of the database, it is possible to do a quick incremental database retrieval and ensure that the status of a group of files is up to date in the local database. The IPC mechanism provides a quick and efficient means of sending this information between machines. In addition, the programmer is guaranteed that even if his local database is not completely up to date, no operation will be permitted which will leave the system in an inconsistent state, since all requests are mediated by the database controller.

There were a number of reasons why the system was designed with a single database controller, rather than a more general distributed database architecture using a voting or locking protocol.

o The design using a single controller is simpler.

o We could guarantee, to a high order of reliability, that the machine running the controller would always be up. However, we could not guarantee that all or even most of the other machines would be running when some user wished to use the system. We would then have to deal with the problem of partitioning of the database and merging conflicting databases when partitions dissolve.

o We wished the system to be as non-intrusive as possible. Any system involving voting or distributed locking would have required servers on each

machine doing some processing whenever anyone was using SSD. Though the actual amount of processing might be small, paging costs would be visible to users as their machine responded to a request.

o A centralized mechanism would give the fastest response time since it would require the minimum number of interacting machines.

Our experience with this design has been excellent. The response to a request is fairly quick and we have had no problems with inconsistent databases. Interaction with the controller tends to be short and fairly sporadic.

4.2.4 User Interface

There are two styles of user interaction with SSD. One is a tool-oriented interface in which the user specifies a set of files on the command line. For example the user might type

```
Exec> checkout *editdoc*
```

in order to CheckOut all the files associated with the EditDoc editor.

The other interface is through a highly interactive tool which gives the user greater flexibility in specifying the set of files to operate on and permits the user to view the contents of any record in the database. A user is able to fill out a form to match fields in the database record. For example, a user may ask to view all the files which he has checked out. This is implemented as a fairly general query mechanism. Some examples of the types of queries which can be specified are:

o All checked out files.

o All files checked out by some set of people.

o All checked out files except for those checked out by a particular user.

o All files installed within the last week.

o All files which have not been modified for six months.

The ability to query the database has made it much easier to handle dependencies between programmers. It has also enabled us to keep better track of the status of sets of files.

4.2.5 Future Work

Future developments will concentrate on integrating inter-module dependencies within the SSD system. Dependencies can be used to check the legality of Install and BringOver operations in order to ensure that only a consistent set of software is placed on a file system. Dependencies may also be used to automatically determine the actions which need to be taken to bring a set of files into a consistent state, in the style of *Make* on UNIX.

A-29

It will also be valuable to be able to query the database to view the dependencies between modules. If the interface to some widely-used public library changes, the database will be able to tell us what files depend on it and may need to be recompiled or even edited to reflect the changes. Systems like *Make* do not maintain external databases describing interrelations between modules and so provide no mechanism for determining this information.

A further goal may be to integrate some version control system, such as the *Source Code Control System* into the Software State Database framework. It would be possible to integrate a more complex view of the version of a particular module into the system and to implement this view through some other utility such as SCCS. The facilities provided by the two systems complement each other fairly well.

4.3 IPC Monitoring Facility

The IPC monitoring facility is a tool which makes it possible to monitor the interprocess communication (IPC) traffic between the components of a distributed system. This tool is intended to overcome, at least in part, some of the problems that plague the developers of distributed programs. In particular, our work has focused on issues related to distributed system debugging and demonstration.

Distributed System Debugging

The process of debugging a faulty software system can be broken down into three stages: error detection, location and repair.

o Detection — The debugging cycle starts when the software system is observed behaving in a manner which is not in accordance with the system's specification.

o Location — Once it has been determined that the system is behaving anomalously, the next step is to trace the computation backward until a point is found where a module is given a reasonable set of inputs, but produces an unreasonable result. Unfortunately, in many cases, a significant amount of time will have elapsed between the point in a computation when a software error occurs and the time when it is first detected. In the intervening period, information that would have proved useful in back-tracing the computation may have been overwritten.

Even if there is not enough information to back-trace the computation, the developer can still use whatever information is available to formulate a hypothesis as to where the bug is. This hypothesis focuses suspicion on a portion of the software. During future runs of the program, the developer can monitor these suspected portions carefully in the hope of obtaining more information regarding the bug. This additional information may then be used to further reduce the area of software suspected of containing the software error. Assuming that the software error is reproducible, repeated application of the above approach will eventually narrow the area of suspicion to the line or group of lines that contain the error.

o Repair — Once the software error has been found, the next task is to correct the software. The amount of work that this entails will vary

depending on whether the error was the result of a design flaw, interface inconsistency or coding mistake.

The nature of the software system being debugged (i.e. sequential vs. distributed) has little effect on the degree of difficulty involved in stages 1 and 3 of the debugging cycle. The task of error location (stage 2), however, is significantly more difficult for distributed systems than it is for sequential ones. This is due to the existence of multiple, asynchronous processes which are running on multiple processors.

A consequence of multiple processes is that instead of one locus of control, there are now several. Furthermore, the results of the system now depend not only on system input, but also on the relative timing of the processes.

For back–tracing to be effective in locating an error, a computation should be halted as soon as a software error is detected. In a loosely–coupled, distributed system, this type of behavior cannot be achieved. Although it is. possible to immediately halt the processor that detected the error, all of the other processors that are involved in the computation will continue working until an indication of the error condition is propagated to them. As a result of this communication delay, critical information in these processes is likely to be overwritten, thereby making back–tracing that much more difficult.

Distributed System Demonstration

·For many distributed application programs, it is important that the application's users be insulated from the distributed nature of the system. For example, the user of a distributed operating system does not need to know where his files are stored. The distributed operating system takes care of those details for him. While this abstraction layer makes the system easier to use, it frustrates any attempt to demonstrate, via the user interface, the underlying interprocess interactions.

4.3.1 System Design

As discussed above, the developers of distributed application programs are faced with debugging and demonstration problems for which the development tools designed for sequential programs are inadequate. The IPC monitoring facility is intended to remedy some of these inadequacies by providing a way to examine the information that passes between the components of a distributed system. This section describes the initial design of the IPC monitoring facility. The topics discussed include: design considerations, basic mechanisms, and system architecture.

4.3.1.1 Design Considerations

The design of the IPC monitoring facility was influenced by the following set of considerations and goals. Some of the goals conflict and they are therefore described in order of decreasing importance.

1. Support the development of Diamond and other distributed programs which are based on the same underlying interprocess communication mechanism.

Diamond is a loosely—coupled distributed system which may be viewed as a collection of communicating objects. Each object has a type (e.g. document, folder, process) and a unique identifier (UID) which serves to distinguish an object from all others in the system. When one object wishes to communicate with another, the sending object constructs a message and addresses it to the UID of the intended recipient. This message is then given to the local interprocess communication server (IPCServer) for delivery. The IPCServer determines the host where the destination object currently resides and transmits the message to the IPCServer on that host. The IPCServer on the destination host then determines the type of the object to whom the message is addressed. If the intended receiver of the message is a process, the message is delivered directly to the object. Otherwise, the message is delivered to the process which is responsible for managing objects of the indicated type.

In this initial design, the monitoring program receives copies of the messages that are being monitored (passive monitoring). The more complicated task of active monitoring, in which the monitoring program intercepts the interprocess messages being monitored, is not addressed.

Although the IPC monitoring facility has been tailored to support the needs of the Diamond and Cronus[3] distributed systems, many of the ideas and mechanisms discussed below are applicable to any distributed program that is based on a message—oriented interprocess communication facility. The nature of the support that the IPC monitoring facility provides for the development of such programs is explained below in the remaining design considerations.

2. Allow users to examine the behavior of a distributed program from a single work station.

Debugging utilities that were designed as aids for sequential program development are inadequate for monitoring the flow of data and program control as it passes between distributed system components. It is conceivable that a program's behavior could be monitored by running each process in the system through a conventional debugging utility. The output from these debuggers could be displayed on one or more terminals for each of the host computers involved. At best, such an approach would be awkward, inefficient and time consuming.

The IPC monitoring facility shifts the "leg work" of distributed program debugging from the program developers to the computer. The information which is of interest in the distributed system is automatically collected and retrieved for either immediate display or later analysis.

3. Display the distributed system behavior in a way that clearly indicates component interconnections and the relative timing of component interactions.

An understanding of the flow of data and program control is essential in

---

[3]M. Hoffman, W. MacGregor, R. Schontz, R. Thomas, E. Burke and B. Woznick, *Cronus, A Distributed Operating System. Preliminary System/Subsystem Specification*, BBN Report 5260, February 1983.

debugging any program. For distributed programs, the existence of multiple loci of control make it particularly difficult to comprehend the program's behavior.

The IPC monitoring facility makes it easy to visualize the behavior of a distributed system. As shown in Figure 2, the components of a distributed system are represented graphically. Host computers are represented by large circles which enclose process objects. Processes are represented by ovals and connections between processes and hosts are indicated by lines. Messages from one process to another are represented by small circles and are shown travelling along the intercomponent connections.

4. Instead of monitoring all interprocess messages, allow users to specify the set of messages that are of interest.

When debugging a piece of software, it is important to focus attention on the region of the program or set of interprocess interactions that is suspected of being faulty. Any information pertaining to portions of the system that are believed to be working correctly is of little value. Indeed, such information may have a deleterious effect by distracting attention from the parts of the program which are suspect.

Through the use of filters, the IPC monitoring facility allows a user to specify the subset of the total message traffic that is to be monitored. A message is monitored only if the contents of the message match at least one of the specified filters.

5. Support the monitoring of transactions.

Many of the interactions between Diamond's distributed components follow a Client/Manager paradigm. One process, acting as a client, invokes an operation on an object. This is accomplished by sending the object a message which contains the name of the operation to be performed and the input arguments for the operation. This message is delivered to the process responsible for managing the object. The manager performs the operation and sends a message containing the result arguments back to the client. Note that in the course of performing an operation, a manager may, in turn, invoke other operations, thereby behaving like a client toward some other manager. The initial operation and all of its suboperations are identified by a transaction ID.

For both debugging and demonstration purposes, it is useful to think of monitoring in terms of transactions rather than individual interprocess messages. Accordingly, the IPC monitoring facility allows users to specify transaction filters. If the contents of a message matches one of these transaction filters, the matched message and all subsequent messages that have the same transaction ID as the originally matched message will be monitored.

6. Allow users to control the level of detail at which information about the distributed system is displayed.

Interprocess messages in Diamond have two parts, a header portion and a data portion. The header portion contains information related to message transport (e.g. source UID, destination UID, data offset and data size) and is encoded according to an IPC peer-to-peer protocol. The data portion of a

A-33

message is of variable length and is encoded as name-value pairs according to the OP protocol.[4] Each value in a name-value pair has an associated type. These types are either simple (e.g. integer, boolean) or structured (e.g. array, record). Furthermore, the elements of a structured type may themselves be either simple or structured types.

Both portions of a message are potentially of interest in the debugging or demonstration of a distributed system. Accordingly, the IPC monitoring facility allows users to examine all of the information that a message contains. In order to avoid overwhelming users with unwanted information, however, the data that is contained in the message is interactively displayed in response to "more detail" and "less detail" operations. This is patterned after the mechanism for displaying variables in the Jade debugger. When a user asks for more detail on a message, the names and values of the top level fields in the message are shown. If additional information exists for any of the displayed fields, it may be obtained by positioning the mouse cursor over that field and asking again for more detail. Less detail, returns the user to a display of the data at the next higher level. In this manner, the user can examine the contents of the message at any level of detail. Furthermore, since knowledge of the IPC peer-to-peer and the OP protocols is built into the IPC monitoring facility, the message fields are formatted appropriately when they are displayed.

7. For messages that are being monitored, record time stamps for key events along the message transport path.

The recording of time stamps along the message transport path has two uses. First, this information makes the detection of race conditions possible in that the relative arrival times for two messages that are sent to the same process may be determined. Secondly, the differences between the time stamps taken along the message path are indicative of the performance of the distributed system.

As shown in Figure 3, the IPC monitoring facility records time stamps at six points along the message transport path. The events associated with these six time stamps, T1 through T6, are as follows.

T1   The sending process sends the message.

T2   The local IPCServer receives the message.

T3   The local IPCServer sends the message to the remote IPCServer.

T4   The remote IPCServer receives the message.

T5   The remote IPCServer sends the message to its intended recipient.

T6   The intended recipient receives the message.

---

[4] A description of this protocol is given in Section 4.2 of *Research in Distributed Personal Computer-Based Information Systems. Semi-Annual Technical Report No. 4. BBN Report 5722, September 1984.*

8. Permit the display of interactions between the components of a distributed application while the application is executing.

As the monitored application executes, monitoring information is collected and sent back to the monitoring host. Since the application execution and the monitoring information retrieval proceed in parallel, the behavior of the application can be displayed in (near) real time. Furthermore, the monitoring information is recorded and may be replayed. This makes it possible to study the application by repeatedly displaying its behavior.

9. Minimize the "costs" associated with interprocess message monitoring.

The use of the IPC monitoring facility adds a certain amount of overhead to the running of a distributed application. The matching of messages against filters and the transport of monitoring information back to the monitoring host all serve to degrade the performance of the distributed application being monitored. Care has been taken throughout the design of the IPC monitoring facility to limit such "costs".

4.3.1.2 Basic Mechanisms

At a high level, the function of the IPC monitoring facility is simply the collection, retrieval and presentation of monitoring data. These three tasks and the work that they entail are described in greater detail below.

Data Collection

Data collection in the IPC monitoring facility is supported by a filter mechanism which allows a user to specify the subset of the total message traffic that is to be monitored.[5]

Once a filter has been specified, it is assigned a filter ID and distributed to all of the IPCServer processes in the distributed system. The filter ID consists of the internet address of the host on which the filter was specified and a sequence number which distinguishes the filter from all others that have been specified on that host. By distributing these filters, we ensure that each IPCServer has a complete copy of the filter data base. Consequently, the matching of interprocess messages against those filters does not require any interhost communication.

Although the task of matching messages against the filter data base does not require any interhost communication, it is still a fairly "expensive" operation which must be performed for every interprocess message that is sent. To avoid slowing down the IPC facility on hosts which we are not interested in monitoring, a mechanism is provided for enabling or disabling the monitoring facility on a host. Hosts which are not enabled for monitoring do not maintain a copy of the filter data base or attempt to perform any message matching. To actually monitor an interprocess message, therefore, both of the following conditions must hold.

o The message's source and destination hosts are both enabled for monitoring.
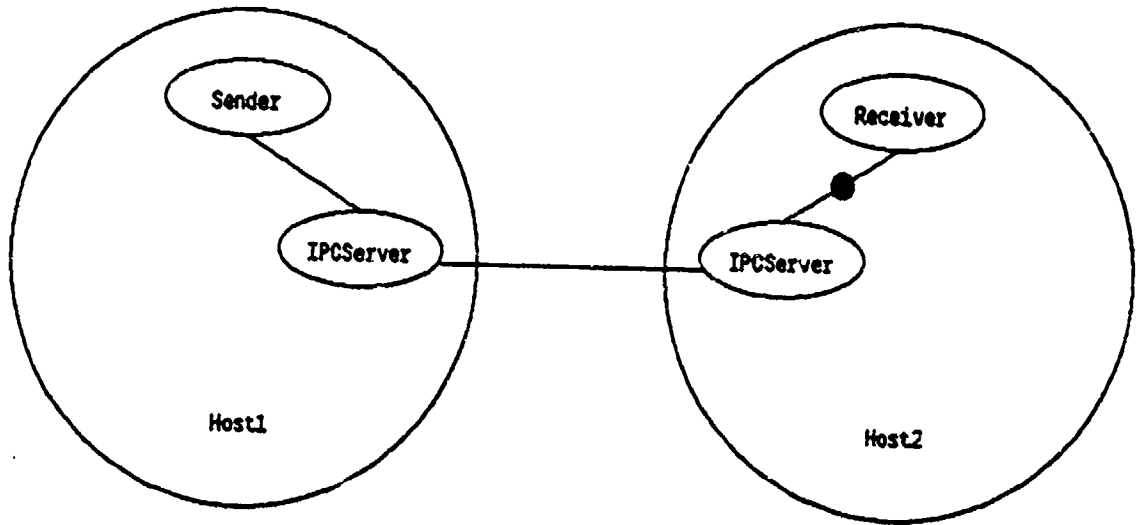
---

[5]Ibid. Section 4.4.1

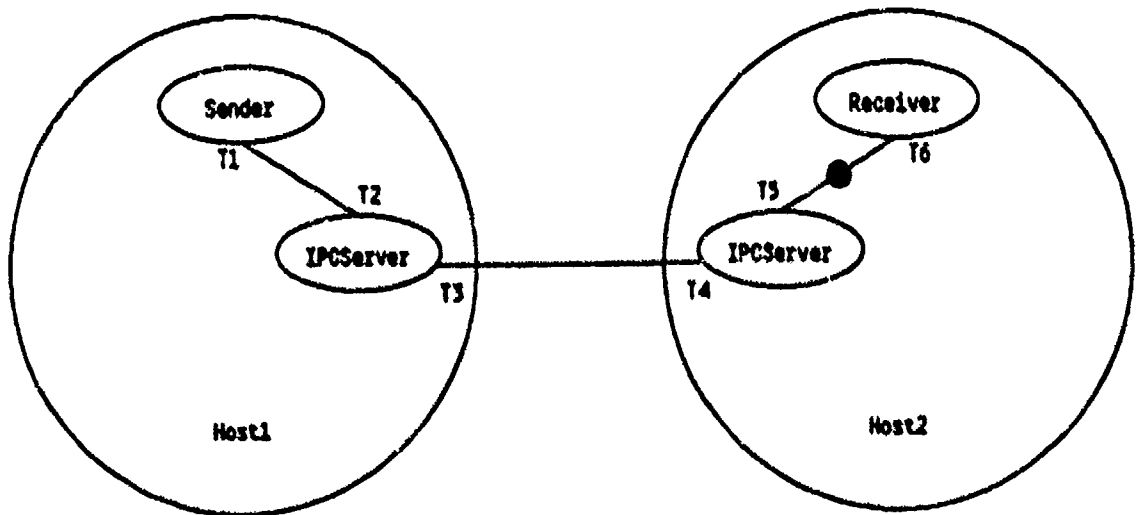Figure 2. The graphical display of an interprocess message

Figure 3. The time stamps along an interprocess message path

o The message matches at least one of the filters in the filter data base.

Whenever an interprocess message is sent from a host for which monitoring is enabled, the message is matched against the filter data base. If one or more matches are found, the header portion of the message is modified to indicate that the message is being monitored. Monitoring a message entails collecting information at several points along the path of the message and sending this information back to the monitoring host(s). The information that is collected includes: a copy of the interprocess message, the name and process ID of the process that sent the message, the name and process ID of the process that received the message, and time stamps that are associated with events along the message's path.

Data Retrieval

Once the message information has been collected at the remote hosts, it must be sent to the host that is running the monitoring program. This program coordinates and displays the incoming monitoring information from all of the hosts in the distributed system.

The monitoring information for a message is sent to the monitoring program in the form of three meta-messages. Each of these meta-messages corresponds to a point along the message path and contains information that was available at that point. The first meta-message contains a copy of the contents of the message, the name and process ID of the sending process and the first three time stamps for the message (T1 through T3 in Figure 3. This meta-message is sent when the original message leaves the IPCServer on the sender's host.

The second meta-message contains the name and process ID of the destination process, and the fourth and fifth time stamps for the message (T4 and T5 in Figure 3. This meta-message is sent when the original message leaves the IPCServer on the receiver's host. Note that if the sending and receiving processes for the message are both on the same host, the first two meta-messages would be sent at the same time. Rather than send two separate messages, the information is instead merged into a single meta-message.

The third meta-message contains the last time stamp for the monitored message and is sent when the destination process finally receives the message. For the case where the receiving process happens to be the IPCServer itself, the information from the second and third meta-messages is combined into a single meta-message.

In addition to the information described above, each meta-message also contains a Message ID which identifies the original message that corresponds to the monitoring information contained in the meta-message. This Message ID is the same for each of the three meta-messages and is used to collate them.

Data Presentation

The last phase of the monitoring facility is responsible for the merging, sequencing and display of incoming monitoring information. As meta-messages arrive at the monitoring host, they are written to a log file. This log file is then used to drive a program which displays the behavior of the distributed application that is being monitored.

As a result of interhost communication delays, the order in which meta—messages arrive at the monitoring host typically differs somewhat from the sending order of the original messages. One of our stated goals, however, is to display messages in a way that clearly indicates the relative order in which the messages were sent. While it is conceivable that the messages could be sequenced according to their time stamps, in practice the synchronization of clocks in a distributed system is a difficult problem. Rather than require a "global physical clock", the IPC monitoring facility instead uses a logical clock to order messages.

The logical clock mechanism is essentially a way to assign a number to a message, where the number is thought of as the time that the message was sent. This mechanism ensures that if message B was sent as a result of message A, then the clock value that is associated with message A will be less than that which is associated with message B. The implementation of logical clocks is straightforward and may be summarized by the following two rules:

1. Before a message is sent, the logical clock on the sending host ticks. This new clock value is the logical time stamp for the message that is being sent and is included in the header portion of the message.

2. When a message is received, the logical clock on the receiving host is adjusted so that the clock value is equal to the maximum of the former clock value and one more than the logical time stamp on the received message.

   Clock := Max(Clock, ReceivedTimeStamp + 1)

Logical clocks ensure that no matter what order the meta—messages arrive at the monitoring host, it is possible to sequence them so that causal relationships between messages are preserved. Once the meta—messages have been merged and sequenced, they may then be displayed by the monitoring program.

4.3.2 System Architecture

In order to monitor distributed application programs, the IPC monitoring facility has to itself be a distributed system. For every host that is engaged in the display of distributed system behavior, there is one IPCMonitor process and one MetaMsg Manager process. Furthermore, those hosts that have been enabled for monitoring are also each running a MonitorData manager process. Enabling a host for monitoring means that as interprocess messages are sent, they are matched against the set of monitoring filters. If a match is found, the monitoring information is collected, sent back to the monitoring host that specified the filter, and displayed. For example, Figure 4 shows a distributed system where distributed application programs are being monitored on host H1. Hosts H1 and H2 have been enabled for monitoring but host H3 has not. The processes that comprise the IPC monitoring facility and the manner in which they cooperate with one another are described in greater detail below

4.3.2.1 IPCMonitor

The IPCMonitor process is the user interface to the IPC monitoring facility. Through this interface, a user may add and remove monitoring filters; enable or disable monitoring on the distributed host computers, and control the display of monitoring information that is being collected and retrieved by the other components of the IPC monitoring facility.
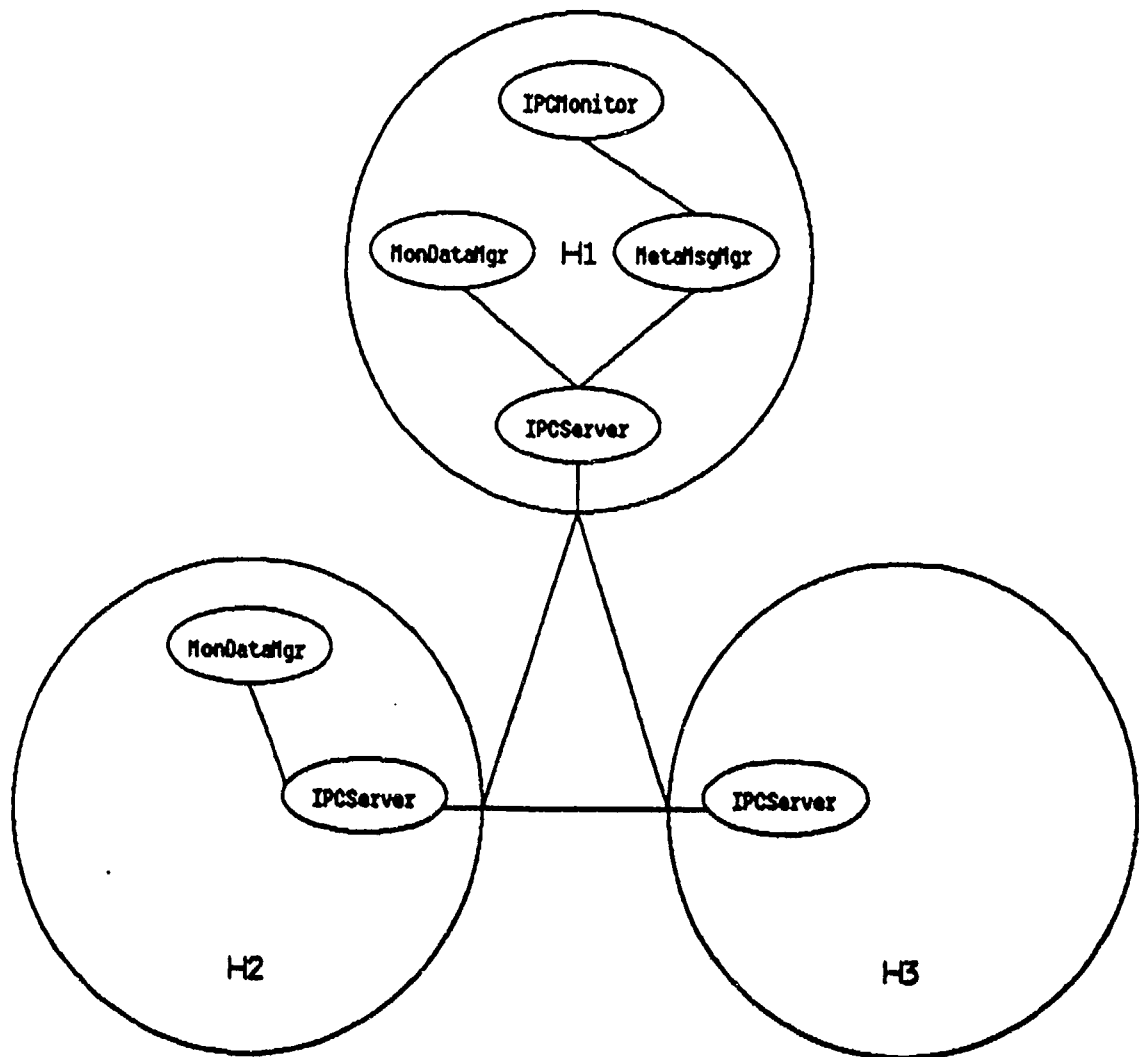
A-38

Figure 4. The component processes of the monitoring facility

A user is allowed to vary the speed at which information about the program being monitored is presented. This capability makes it possible to slow down or even completely freeze the message traffic display so that the content of the displayed messages may be examined. Slowing down the display of information, however, does nothing to slow down the rate at which new monitoring information is arriving. To avoid the backlog of unprocessed monitoring information that would otherwise result, a separate process, the MetaMsg manager, is used to process incoming meta-messages.

Under this approach, when the IPCMonitor is ready to display more information, it just asks the MetaMsg manager for the next chunk of monitoring data.

4.3.2.2 MetaMsg Manager

In addition to processing incoming meta-messages, the MetaMsg manager also records the message information in a log file so that it is possible to repeatedly display the monitored behavior of a system.

The MetaMsg manager also acts as a distributor for any new filters or changes to existing ones. The MetaMsg manager maintains a table of all of the hosts in the distributed system that are currently enabled for monitoring. Whenever, the user adds or changes a filter, the IPCMonitor passes the filter request to the MetaMsg manager. This process then transmits the request to all of the known MonitorData managers in the configuration.

Finally, the MetaMsg manager is also responsible for exercising a degree of global control over the logical clock mechanism described in Section 4.3.1.2. Although logical clocks ensure that no matter what order messages are received by the monitoring facility, it is possible to order them so that causal relationships are preserved, they do not provide a way of determining when all of the messages with logical time stamps that are less than some logical clock value have been received. This is needed in order to determine when it is "safe" to commit the actions required in displaying a message. The MetaMsg manager remedies this deficiency by periodically polling all of the MonitorData managers for the earliest possible logical time stamp of sent messages that may still be in transit. By taking the minimum of these local earliest possible time stamp values, a global earliest possible time stamp (GEPTS) is obtained. All display actions resulting from time stamps that are less than the GEPTS value may be committed.

4.3.2.3 MonitorData Manager

When a host is enabled for monitoring, the IPCServer creates the MonitorData manager which is used for communicating between the IPCServer and MetaMsg manager processes. This is needed because the IPCServer cannot send and receive messages in the same way that a typical client process does. Accordingly, the MonitorData manager process is used as an intermediary.

When the IPCServer sends a message to the MetaMsg manager, the message is first passed to the MonitorData manager process using an intrahost message transport mechanism. The MonitorData manager, which is a normal client process, then sends the message to the MetaMsg manager using the interhost IPC facility. Messages from the MetaMsg manager to the IPCServer are sent in a similar manner -- first to the MonitorData manager and then to the IPCServer.

In addition to the IPCServer's main responsibility of transporting interprocess messages, it also maintains the logical clock and checks the messages that it sends against the set of monitoring filters. If a match is detected, a meta-message is constructed which is then sent to the appropriate MetaMsg manager.

4.3.3 Current Status

During this reporting period, we completed the initial design of the IPC monitoring facility as described above and began work on the implementation of some of its low-level mechanisms.

The existing IPC facility has been modified to provide support for monitoring. This entailed modification of the IPC peer-to-peer protocol, implementing the logical clock and filter matching mechanisms, and providing an intrahost communication path between the IPCServer and MonitorData manager processes.

Although the MonitorData manager portion of the IPC monitoring facility has been implemented, work on the MetaMsg manager and IPCMonitor implementations is just beginning.

5. JERICHO INTERLISP

The objective of the Jericho Interlisp task is to port and extend the Interlisp programming language and environment to BBN's Jericho personal computer. Interlisp is one of the two major dialects of the LISP programming language which underlies most research in Artificial Intelligence. This task services two principal goals. First, it provides the development environment in which other DARPA supported research in Artificial Intelligence proceeds at BBN, most notably the work in natural language understanding and knowledge representation. Second, this task builds the foundation for the ALEPH component of this project which is to explore novel programming techniques and tools.

The task of porting Interlisp–10 from a mainframe to a personal computer involves three categories of effort: 1) porting the initial system, 2) extending it to accommodate the functional capabilities of the hardware, and 3) system maintenance. In previous reporting periods, we have addressed each of these areas. In the current period, we continued system maintenance activities; in particular, we improved the reference counting garbage collector, we added a CATCH, THROW, UNWIND–PROTECT facility, and we worked toward incorporating a multiple–process capability in anticipation of interoperability.

5.1 Garbage Collection

During the current reporting period, we extended the reference counting garbage collector to re–use variable length data types, specifically lists and arrays. In addition, we improved its performance.

When the reference counting garbage collector finds a variable length datum to be freed, it marks the free region in a bit array which has one bit for each word of virtual memory. This array is the same one that is used by the compacting garbage collector to mark used data (see the previous report for more detail).

The allocator for variable length types initially assigns a contiguous segment of memory for the type. A segment is currently 8K words. When a segment becomes full, the allocator searches the regions of the bit array corresponding to segments of the desired type for a contiguous free chunk. The minimum size of the chunk is the maximum of the size required for the current allocation and a user settable parameter \FREMIN. The value of \FREMIN is initially 100. If a chunk is found, it is marked used in the bit array and allocation proceeds from the chunk. If no chunk is found, a new segment is assigned. To avoid searching the same sections of the bit array, we start each search where the last one left off. The starting point is reset after every garbage collection.

We initially used a slightly different me..iod for reusing the free regions. At every garbage collection we built a chain throu h the free chunks with the first word of each chunk pointing to the next one. This r.ethod has terrible paging performance because it requires referencing the list or a ray space as well as the bit array. Consequently, we abandoned it in favor of the s arch technique.

This technique for variable length items required some modifications to the compacting garbage collector. Since the compacting garbage collector does not compact arrays, it is important to let the allocator know what array space is free. This is accomplished by initially marking all array space as free. When an array that is used is encountered, its extent is then marked as used.

The above scheme uses microcode primitives to set or clear a number of bits and to find the next zero or one in a region of an array. These primitives are useful for other things as well so they have been made available to the user as new op-codes.

We have improved the performance of the reference counting garbage collector by implementing more of it in microcode. The portions now in microcode are the atom hash table scan (reported previously), the reference table scan to clear the stack bits, and some assistance for the reference table scan for zero count entries to be feed. This last scan searches from the last entry it found (initially 0) returning the next pointer to be freed or NIL if none is found. We also added a microcode operation to add a datum to the free list for its type.

The speed of the reference counting garbage is quite good. The time required to add and delete entries from the reference table is not measurable for any computation that we have seen. The fixed overhead for a garbage collection in the case where nothing is found to free is less than 0.25 seconds.

5.2 Catch, Throw, and Unwind-Protect

Interlisp provides the basic non-local exit facilities RETTO and ERROR! as well as several others based on them. RETTO causes a return to a specified stack frame while ERROR! returns to the nearest stack frame named ERRORSET or to the top level if no such frame exists.

Common Lisp provides similar abilities with CATCH and THROW. Because CATCH and THROW are more versatile than the ERRORSET, ERROR! combination, we have included these facilities in Interlisp Jericho.

A problem with non-local exit facilities is that sometimes a compution wishes to assure that certain cleanup operations (such as closing files) are performed even if the computation is exited non-locally. Interlisp provides a group of macros called RESETLST, RESETFORM, RESETSAVE etc. which attempt to address this problem. However, there are several deficiencies. The resulting code is hard to read, the cleanup form is computed at run time and thus is not compiled, and in the particular case of a non-local exit to the top level, the cleanup form is evaluated in the wrong environment; namely, at the top-level rather than in the environment in which it appears.

The Common Lisp UNWIND-PROTECT is a clearer and better method so we have included it in Interlisp Jericho. (UNWIND-PROTECT protected-form |cleanup-form|*) evaluates protected-form and the cleanup-forms in the correct environment whether the UNWIND-PROTECT is exited normally or via a THROW to a containing CATCH. The cleanup forms are also evaluated during RESET or control-D, which are effectively throws to the top level, and during ERROR!, which is effectively a throw to the most recent ERRORSET.

The Interlisp non-local exit RETTO is sometimes used to abort a computation but is also used for coroutines and generators where the abandoned computation may be resumed. In that case, the cleanup-forms probably should not be evaluated so RETTO and its derivatives have not been changed. We have added two new functions, RETTO&UNWIND and RETFROM&UNWIND, which do evaluate cleanup-forms for UNWIND-PROTECTs between their invocation and the destination frames. In the current implementation, if the destination frame is not in the current stack environment, these functions will unwind to the top-level. This should probably be changed to cause an error.

5.3 Multiple Process Capability

Our ultimate goal under the concept of interoperability is to provide for a multiple process capability for both Lisp and Pascal simultaneously. Our original design strategy called for a microcode kernel which would implement the basic process mechanism to be shared by both language environments. But an additional constraint forced us to reconsider this approach. That constraint is imposed by our need to keep Interlisp Jericho compatible with Interlisp-D.

Under this constraint, and after looking at the Interlisp-D multiple process facility, we decided that sharing the microcode kernel was an untenable way of achieving Interlisp-D compatible multiple processes in Interlisp Jericho. This was primarily because the kernel embodies a preemptive form of scheduling whereas Interlisp-D is based on a non-preemptive scheme. Consequently, we changed our design to yield a compromise between these two forms of scheduling.

The result is that we plan to first bring up a multiple process capability in Interlisp Jericho by adapting and integrating the Interlisp-D process package. Later, we will execute this version of Interlisp Jericho as a single process among the Pascal preemptive processes controlled by the microcode kernel. We will then have multiple processes running for both languages simultaneously and we will maintain compatibility with Interlisp-D.

During this reporting period, we have begun the effort to adapt the Interlisp-D process package. Since multiple process code so heavily impacts all of the other system code, we have also taken this time as an opportunity to update the Interlisp Jericho system with the newest version of Interlisp-D. We are making progress toward this goal but still have much left to do. In particular, the input/output software has been dramatically affected by the transition to multiple processes as has the interrupt system. We fully expect to resolve these problems in the coming months.

6. ALEPH

The goal of the ALEPH component of the project is to conceive and test new ideas and tools which can aid the programmer in his task. We expect that the capabilities of a personal computer such as the Jericho can offer new opportunities in this area, particularly the high-resolution bitmapped display.

In this reporting period, we continued our two major thrusts which were documented in the previous report: Content Addressed Documentation and Programming Tools. In the first category, we investigated and implemented the dynamic catalogue idea which we call the Interlisp Advertiser. In the second category, we discuss four tools we developed: the Directory Browser, File Comparison Presentation, the Code Presenter, and the Graphical Debugger.

6.1 Content Addressed Documentation

Users of sophisticated programming environments (such as LISP) often confront problems stemming from the breadth and scope of the procedural capabilities of the system. One of these problems is finding information about needed functional capabilities. Sometimes these capabilities are known to exist but their name is unknown or can not be recalled; often, however, users do not know if the capability is or isn't available in the system and they must simply browse and "discover" it. As described in our previous semiannual report, the solution that we have been exploring consists of "advertising" a functional capability by showing in a small "movie screen" or advertising window a short "movie strip" of what the advertised capability can do. During this reporting period we have begun developing a tool, which we call "The Interlisp Advertiser", that is based on this idea.

6.1.1 The Interlisp Advertiser

The Interlisp Advertiser can be considered an advertising catalog, in which the various features and system capabilities are shown in advertising windows. Much in the way in which commercial catalogs are organized, our advertising windows can be grouped thematically and displayed simultaneously in a single page. As with a paper catalog, a user can converge on a relevant section by consulting a thematic Table of Contents, an Index, or simply by flipping pages. In contrast with hard-copy catalogs, however, users of our tool can derive a sense of action from the animation in the advertising windows, and can obtain relevant information (e.g.: the name of the advertised function or feature) by pointing to one.

Having found the information, another problem that often arises is how to use it, after suitable modifications, in the user's particular situation and context. A solution that we have just begun to explore is to provide a "hands on" workshop environment that facilitates modification and experimentation of an existing functional capability to adapt it to the user's requirements, and that makes it easy for the user to incorporate the modified capability into his own software.

We shall next describe how the system appears to its users. As an example we

Preceding Page Blank

shall use an "ad hoc" page of this kind of "advertising catalog"; this is a page that we have constructed to conveniently illustrate the various features of the system but which lacks thematic unity and, as a page, would therefore not be included as a part of the Advertiser.

In this "ad hoc" page, which appears in the following figures, we have included 8 advertising windows. Each advertising window shows an animated sequence of what a particular function or feature of the Interlisp display package does. Thus, the SCROLLing window shows how to scroll text printed in a window, the textPOSITION window shows how to specify where text is to be printed, and the SOURCE&OPERAT. window shows how to combine background bits and displayed bits to obtain a variety of visual effects. The other windows show how to adjust the shape of a window, change its background texture, select fonts, draw ellipses and compute region intersection operations. The bottom of the page is occupied by a Help window, which appears in all the system's pages.
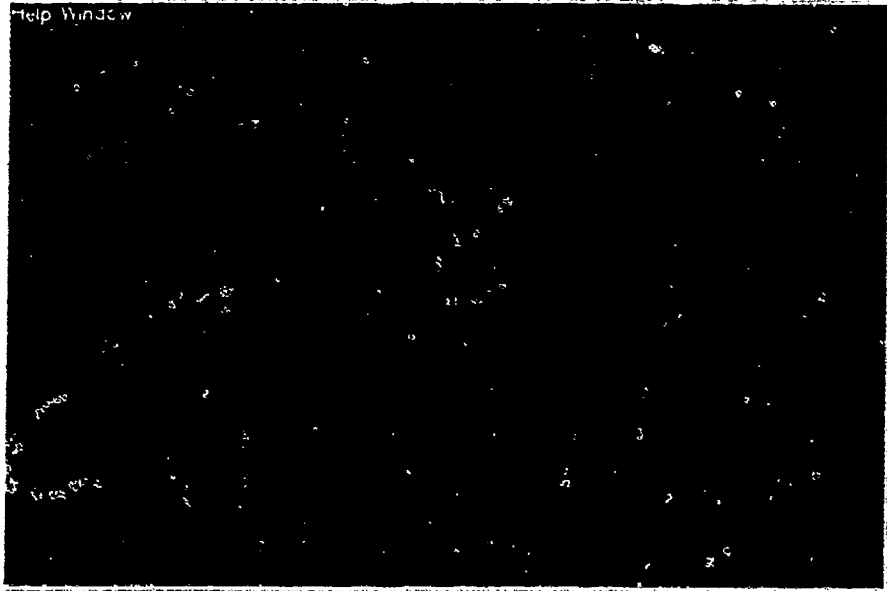
In figures 5 through 9 we have tried to capture what happens during the animation sequences (albeit with marginal success). Each figure depicts the screen as it appears after each window has played one frame of the animation. Thus, during the first frame a cursor moves leftward on the SCROLLing window to simulate moving the mouse so as to touch the left border (a smudge is still visible in figure 6, overlapping the "p" in "properly"), and in the textPOSITION window, an arrow flashes indicating where the next line of text (the "the beginning" line, also in figure 6) is going to appear. Similar demonstrative actions take place in between the images in figures 6 through 9.

The animated sequences in each window can all "play" simultaneously as shown, or one window at a time at the user's discretion. Depressing the left button on the mouse (see the BUTTONS window on the upper left hand corner) causes a COMMAND menu to appear, with which the user can control the way information is displayed. Pointing to a window with the mouse and depressing the center button summons a HELP menu with which the user can obtain specific information about the advertised feature.

In figure 10 we have placed the mouse over the FONTS window, and we have depressed the middle button, summoning the Help menu. With it, the user can request information on relevant function (FNS) or variable (TERMS) names, can ask to see how what he has just seen was done (HOW?), can ask for an exemplar of usage (EXAMPLE), or can ask for a demo in which he can exert control over the actions advertised or explore the advertised function's capabilities (DEMO). In the figure we have selected the FNS menu item. The relevant functions (i.e.: procedures available in Interlisp) are listed in the Help Window. Figure 11 shows what happens if someone wanted to know how the third ellipse in the DRAWingELLIPSEs window was generated: pointing to the window with the mouse and selecting the HOW? menu item produces a printout of the code that was used to generate the DRAWingELLIPSEs display, with the portion responsible for that third ellipse highlighted. Finally, figure 12 shows an example of the "hands on" exploration facility: Depressing the Help button in the SOURCE&OPERAT. window and selecting the DEMO item causes the system to show the kind of visual effect that would be created if the user chose the INVERT source type and the PAINT operation. Like for the HOW? menu item, HOWDEMO? shows how the DEMO works, the item is shaded, and users are cautioned appropriately if they select it, until the demo is actually seen.

SCROLLing

(- Sometimes
windows are
too small to
display all
the text that
is printed on
them. In these
cases, if the
window is set

text POSITION

Changing

FONTS

This is the
GACHA 12 BOLD
font

SOURCE&OPERAT.

The character
bitmaps
REPLACE
the texture bits.

SHAPEing

(Adjusting, or
changing, the
form or
shape of a
window.)

DRAWingELLIPSEs

Drawing ellipses.

INTERSECTion

TEXTURE

Help Window

Good afternoon, Mario.
    (NIL)

61←ADVERTISE›

62←

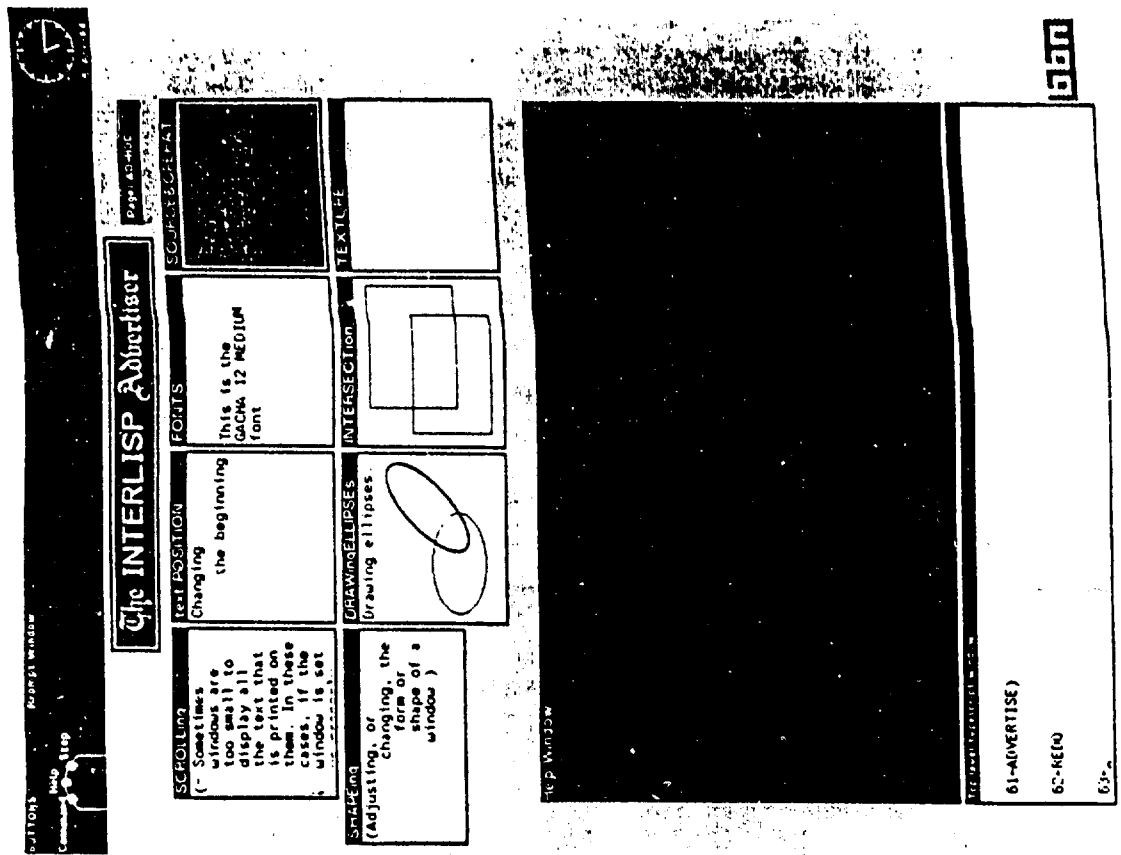Figure 5. AD hoc page of the Interlisp Advertiser frame 1

A-49

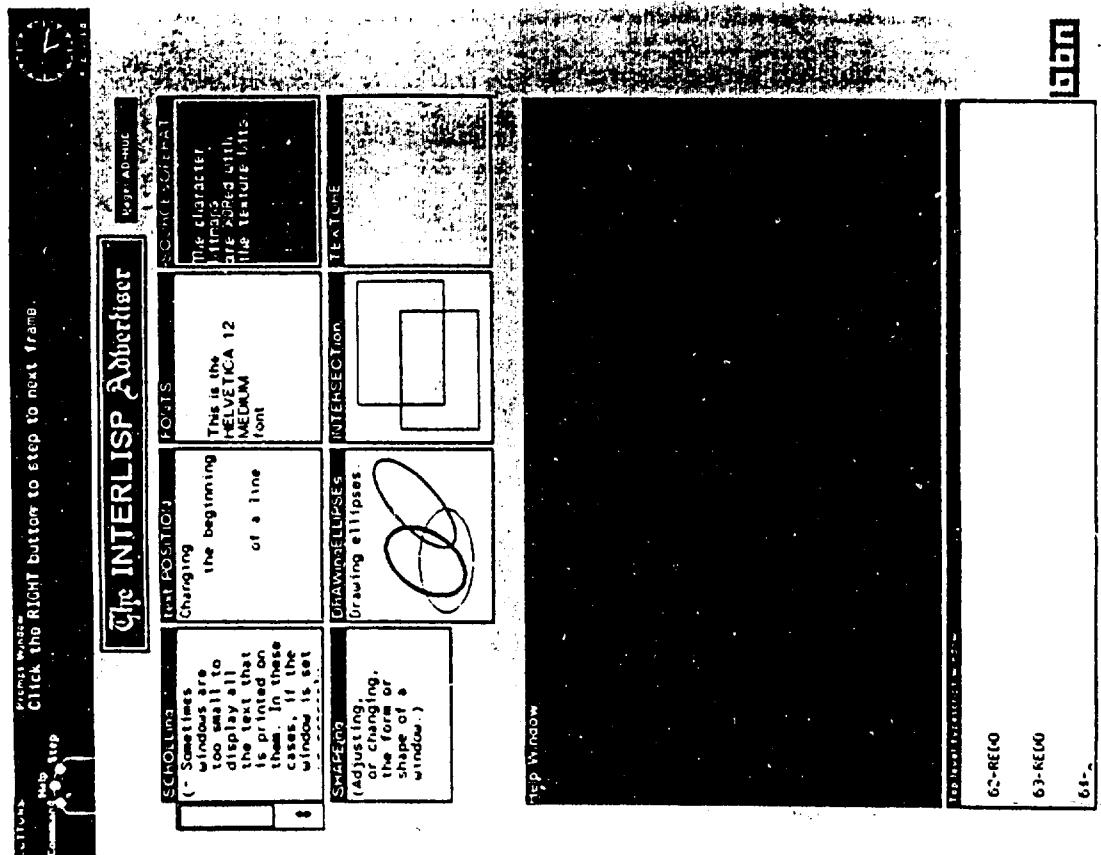Figure 6. AD hoc page of the Interlisp Advertiser frame 2
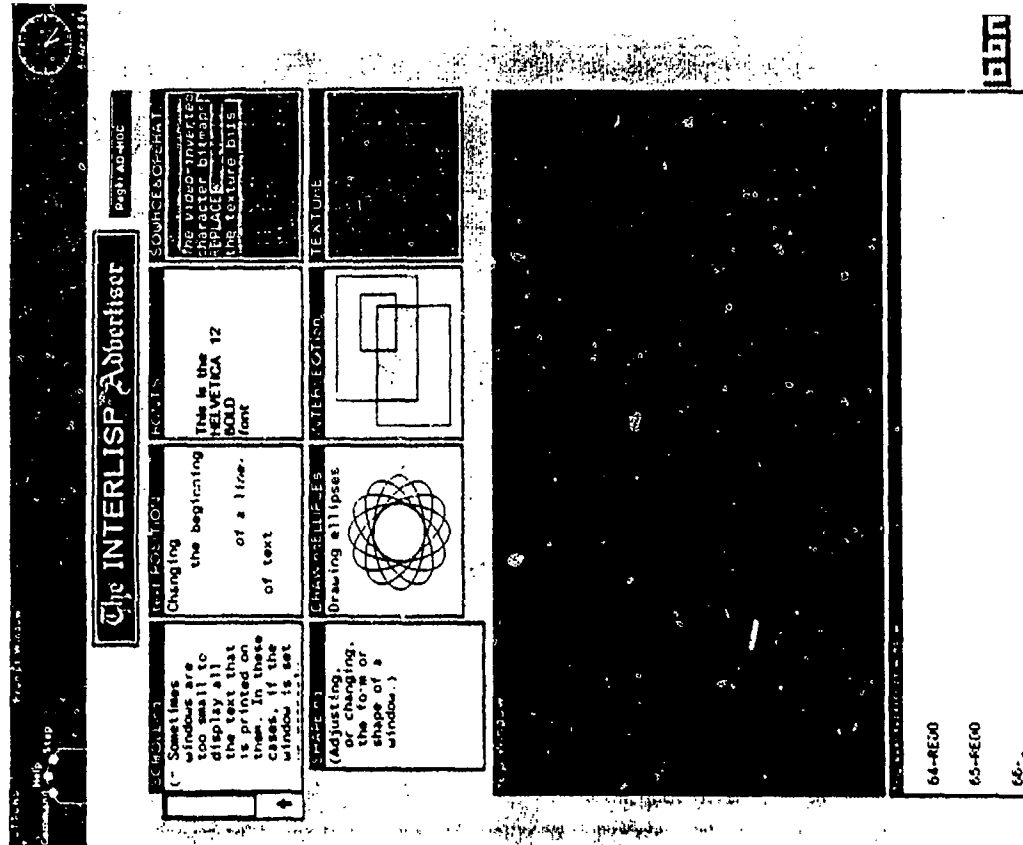

Figure 7. AD hoc page of the Interlisp Advertiser frame 3

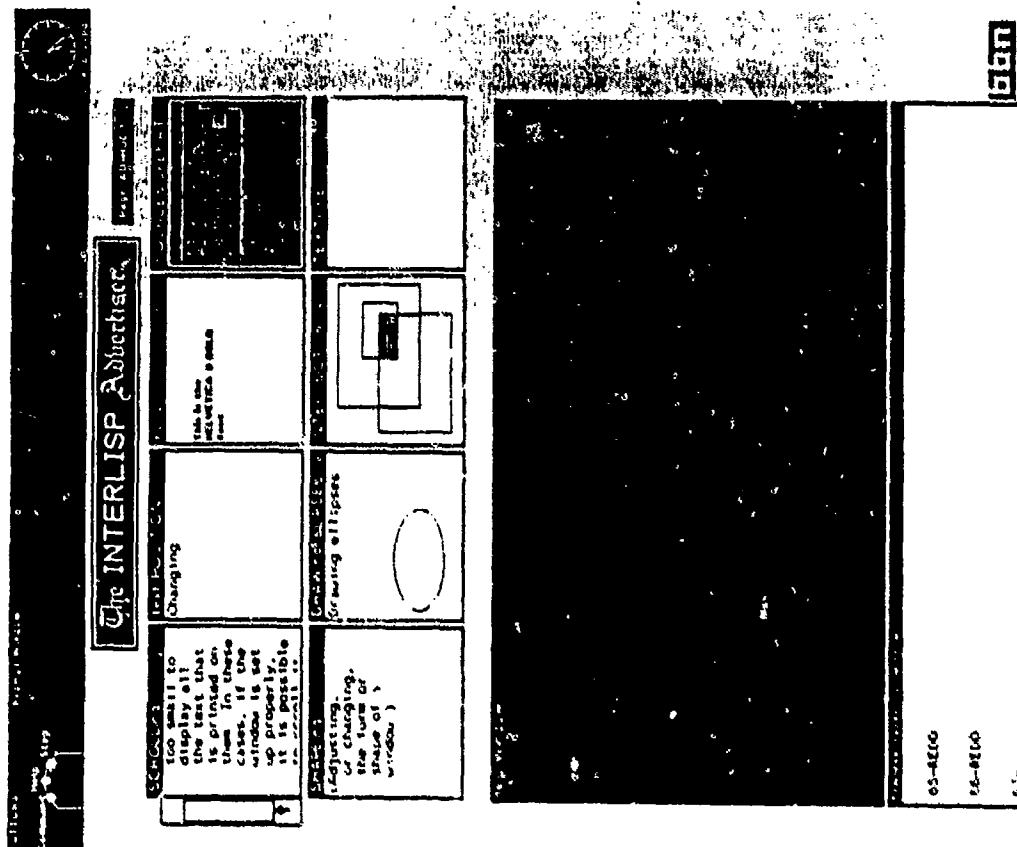Figure 8.   AD hoc page of the Interlisp Advertiser frame 4

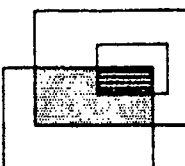Figure 9.   AD hoc page of the Interlisp Advertiser frame 5

BUTTONS   Prompt Window

Command   Help   Step   Will show the names of relevant INTERLISP functions.

The INTERLISP Advertiser   Page: AD-HOC

SCROLLing
too small to
display all
the text that
is printed on
them. In these
cases, if the
window is set
up properly,
it is possible
to scroll it

text POSITION
Changing

FONTS
This is the
HELVETICA 8
font

Help Menu
FNS
TERMS
HOW?
EXAMPLE
DEMO
HOWDEMO?

SOURCE & OPERAT.

SHAPEing
(Adjusting,
or changing,
the form or
shape of a
window.)

DRAWingELLIPSEs
Drawing ellipses.

INTERSECTion

TEXTURE

Help Window

Relevant Functions are
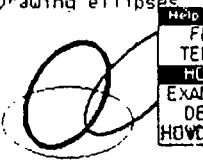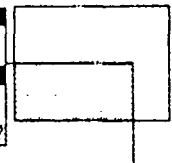
(DSPFONT FONTCREATE FONTCOPY FLINTP FONTPROP)

68-
68-
68-
68-interrupted b

(GETMOUSESTATE broken)
68

bbn

Figure 10. Obtaining Help: Relevant Functions for the FONTS window

BUTTONS   Prompt Window

Command Help Step   Shows this window's advertising function, highlighting the
                       part that is responsible for the current frame

5-Apr-84

The INTERLISP Advertiser

Page: AD-HOC

SCROLLing	text POSITION	FONTS	SOURCE&OPERAT.
(- Sometimes windows are too small to display all the text that is printed on them. In these cases, if the window is set	Changing the beginning of a line	This is the HELVETICA 12 MEDIUM font	The character bitmaps are XORed with the texture bits.

SHAPEing	DRAWingELLIPSEs	INTERSECTion	TEXTURE
(Adjusting, or changing, the form or shape of a window)	Drawing ellipses		

Help Menu
FNS
TERMS
HOW?
EXAMPLE
DEMO
HOWDEMO?

Help Window

```
(ADV*DRAWINGELLIPSES
  (LAMBDA (COROUTPTR CALLPTR)  **COMMENT**
    (PROG NIL
      (DSPOPERATION (QUOTE PAINT)
                    DRAWINGELLIPSE')
      LP  (DSPPRESET DRAWINGELLIPSE')
          (printout DRAWINGELLIPSE3V FONT GACHA12
                    "Drawing ellipses.")
          (DRAWELLIPSE 50 40 25 45 0 (QUOTE (ROUND 1))
                       NIL DRAWINGELLIPSE')
          (RESUME COROUTPTR CALLPTR)
          (DRAWELLIPSE 100 70 20 50 45
                       (QUOTE (ROUND 2))
                       NIL DRAWINGELLIPSE')
          (RESUME COROUTPTR CALLPTR)

          (RESUME COROUTPTR CALLPTR)
          (DSPPRESET DRAWINGELLIPSE')
          (printout DRAWINGELLIPSE3V FONT GACHA12
                    "Drawing ellipses.")
          (* ORIENTATION)
          (QUOTE 18 30 80 90 (CR 150))
          (DRAWROUND 25 70 80 25 90 ORIENTATION
```

Top level typescript window
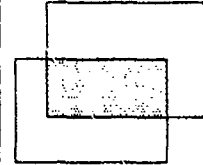
```
68←

58←

←interrupted b
RESUME ABUSE
interrupted belo
```
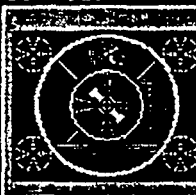
Control-H - GETMOUSESTATE break 1

(GETMOUSESTATE broken)
68 UP
GETMOUSESTATE
(GETMOUSESTATE broken)
70

bbn

Figure 11   Obtaining Help How was the DRAWingELLIPSEs window generated?

A-53

BUTTONS   Prompt Window

Command Help Step   Gives you a chance to practice by letting you interact
                       with the advertised functionality.

5-Apr-84

The INTERLISP Advertiser

Page: AD-HOC

SCROLLIng

(~ Sometimes
windows are
too small to
display all
the text that
is printed on
them. In these
cases, if the
window is set

text POSITION

Changing

    the beginning

    of a line

FONTS

This is the
HELVETICA 12
MEDIUM
font

SOURCE&OPERAT.

Help Menu
FNS
TERMS
HOW?
EXAMPLE
DEMO
HOWDEMO?

SHAPEing

(Adjusting,
or changing,
the form or
shape of a
window.)

DRAWing ELLIPSEs

Drawing ellipses.

INTERSECTion

TEXTURE

Help Window

REPLACE PAINT INVERT ERASE   <- OPERATIONS

INPUT   Depress LEFT button to

INVERT   select SOURCETYPE and OPERATION

MERGE

TEXTURE   To QUIT:

SOURCETYPES

Top level typescript window

REDO ADVERTISE   Control H - BPLUS break 2 .
interrupted belo
interrupted belo   (BPLUS broken)
                   72 *
*1-REDO WINDOWPR
ADV-HELPMENU UNS   (BPLUS broken)
*2-interrupted b   73
                      *

bbn

Figure 12   Obtaining Help. The effects of SOURCETYPEs and OPERATIONs

A-54

6.1.2 Conclusions

Our Interlisp Advertiser derives its leverage from the following key ideas:

o Instead of a mere paper substitute, the screen becomes a dynamic, parallel visual processing interface, i.e.: showing many "mini movies" simultaneously.

o The user's visual reasoning abilities are involved too, not just the verbal ones.

o Uses the mind's visual evocative powers. It elicits visually rather than verbally.

o Uses depictions of events (such as the effects of procedures), rather than descriptions. It is hard to describe space or time sequences verbally. Animated figures work much better.

o It gains additional leverage by acting through the final effect produced, and not on the basis of preconceived ways of achieving the desired effect ("I didn't know one could do it that way").

As a consequence, we believe that our approach opens up new dimensions in system documentation, system development, and on-line performance aids:

1. First of all we have an "active" rather than a "passive" approach to system documentation. Since it relies on figural, dynamic, and ostensive modes of presentation of information, it makes possible to vividly demonstrate system functions instead of just describing them verbally.

2. It makes it much easier for a new user to learn by him/herself. It fosters self-tutoring.

3. It facilitate exploration of choices and instantiation of procedures in the user's domain.

4. It makes it possible to implement pieces of user software by lifting and modifying "advertised" software.

Coping with large amounts of procedural information is not only a problem for system developers. The military end users of the sophisticated systems that will be produced in the next decade will also confront the same problems and will have to depend on self reliance. It is extremely important that such systems have the ability to impart procedural instruction to their users, since it is unrealistic to expect that all users will have always in their heads all the procedural knowledge they may require. Systems incorporating the ideas exemplified in the Interlisp Advertiser should be able to provide such information in a most fruitful way.

6.2 Programming Tools

During this reporting period, we continued to design and implement tools to aid the programmer in his/her activities. Some of the tools are new ideas and others

integrate results of our previous work. All make use of the bitmapped display. Below we describe these tools, which consist of:

o the Directory Browser, which allows inspection and manipulation of the tree structured directory system and the files stored therein

o File Comparison Presentation, which integrates the File Browser with the Heuristic File Comparator to visually display the differences between two files

o the Code Presenter, which provides alternative views of symbolic source code according to the influence of run-time context

o the Graphical Debugger, which provides a graphic interface for program debugging and includes a dynamic graphical tracing facility.

6.2.1 Directory Browser

The Directory Browser, like the File Browser described in our previous report, is an instance of the general notion of browsing. The original notion of browsing was to provide a way of viewing or looking at some object too large and/or too complex to present in its entirety in any one image. A text file, for example, is generally too large to print on a single screen but it is both feasible and useful to show some segment of the file and to allow the user to control which segment is shown. Other objects such as complicated data structures may have an organization best displayed at varying levels of abstraction.

The more current notion of browsing extends the idea to allow manipulation of the object being examined. A browser comes to look much like an editor. Indeed, the generalization of editors to objects other than text and the extension of browsers to allow modification of the object viewed have been two ideas on convergent paths. The emerging idea is one we have called multi-representational editing. The ultimate goal is to provide a variety of presentations for an underlying body of information and, through interactions with any given presentation, change that information. Of course, any change induced through one presentation must be reflected immediately in all other relevant presentations.

Our Directory Browser has been a step along this evolutionary path. It provides different presentations for different facets of a Jericho's file system, and it allows interactions with the files in the system, some of which alter the state of the system. We now will describe the capabilities of this browser and the way a user interacts with it.

The Jericho file system, like that of Unix, is tree-structured. A directory contains entries for files and some of these files may themselves be directories, in other words, directories are special instances of files whose contents are simply file descriptors. One of these directories is distinguished to be the root of the tree-structure and has the name ROOT (it contains an entry for itself as well).

One of the capabilities of the Directory Browser is that it can present a graphical depiction of any subset of branches emanating from the root of the tree representing the directory structure. The user can specify any set of (sub)directories of interest and the browser will determine the appropriate branches to display.

This graph is active in the sense that selecting nodes of the graph cause actions related to the associated directory. One type of selection makes the designated directory the current connected directory (ie. it provides a context for file references which do not specify a directory). This offers the user a very simple way to establish this condition, especially as compared to typing the command when the chosen subdirectory is deep in the hierarchy and a long path name would be required.

A second type of selection opens a scrollable window on the associated directory in which the list of file names appears. Any subset of the file names can be selected and highlighted using the pointing device. A menu can then be used to invoke commands which operate on the selected files. Several different kinds of commands exist.

Some of the commands manipulate the files themselves. These are:

o Browse, which calls the File Browser on the selected file

o Type, which types the selected file in a special window

o Copy, which copies the selected files into another specified directory

o Move, which moves the selected files into another specified directory (this removes the files from the original directory which Copy does not)

o Expunge, which gets rid of the selected files

o Info, which presents relevant information about the selected files such as size in various metrics, dates, and file type

Other commands are related to files which are directories themselves. These are:

o SelectionDirectory, which opens the Directory Browser on the directory file currently selected by the browser

o ParentDirectory, which opens the Directory Browser on the directory which is the parent of the directory currently being displayed by the browser

o ConnectSelection, which makes the connected directory be the directory selected in the browser

o ConnectParent, which makes the connected directory be the directory which is the parent to the directory displayed by the browser

o Connect, which makes the connected directory be the directory displayed by the browser

The remaining commands effect the selections themselves (as opposed to the files represented by the selections). They allow the user to push the current set of selections onto a stack and later pop them back. The user can also clear the current set of selections. These operations are convenient when one is building up a large selection list and the addition of another entry depends on information not currently presented. The user can push the partially collected list of selections, select the questionable file, and then obtain the information necessary for the decision. He/she can then pop back the partially collected list and add or not the file in question.

The File Browser is an extremely convenient tool for the programmer. It provides an easy way to peruse the set of files in a given directory. Any subset can quickly and simply be chosen and a single operation applied to the lot. In addition to this pragmatic benefit, experience with the Directory Browser is beginning to provide support for an important idea; namely, that, contrary to common practice, the better interface is not the one that tries to provide the single best way to achieve something but rather the interface which provides a variety of ways to accomplish something. The user can then choose which mechanism suits him/her best under whatever circumstance. The Directory Browser gives alternative ways to manipulate files in addition to the ways already available, say by typing.

6.2.2 File Comparsion Presentation

In our previous report, we described, among other things, two programming tools: the File Browser and the Heuristic File Comparator. The File Browser provides a way to view a file in a window under user control. The Heuristic File Comparator compares two versions of a file containing program source code in order to determine differences. The comparison is structurally based rather than textually based as has been the practice historically. In this reporting period, we have integrated these two capabilities to provide simultaneous, coordinated browsing of the two file versions including visual annotations of the discovered differences.

Comparison of two files requires the taking of a perspective under which one file is considered the "old" file and one the "new" file. Thus as elements are matched in the structures contained in a file, several cases may obtain: an element appearing only in the new file is an insertion; an element only in the old file is a deletion; elements which are matched in the two files but differ constitute a modification; finally, an element in the old file may be permuted so as to reside in a new position relative to its associated elements. Recall that embedding, another possible type of change, is not handled by the comparator.

In figures 13 and 14, we see old and new versions of the function EQLENGTH, which is a predicate which checks the length of a list. The body of the code is a conditional (COND) with two clauses in the old version and three in the new version. Although either of the first two clauses in the new version could plausibly be matched with the first clause in the old version, the underline of the first clause in the new version indicates insertion of the clause; a mark in the old version appears with a count of one to indicate that a single structure has been inserted at this position in the new version.

The first clause in the old version has thus been matched with the second clause in the new version. Each clause has two elements and the second elements match identically. The first elements are matched and are both lists. In this context, the atoms "ILESSP" and "ZEROP" have been matched and highlighted to indicate modification, the atom "N" in each list is matched without difference, and the atom "1" in the old version has a line through it to indicate deletion – there is a corresponding mark and count in the new version in this position to indicate the deletion.

Finally, down inside the last clause in each version is a list beginning with the atom "NTH." It is a function call with the arguments N and X in the old version and the same arguments but in reverse order in the new version. The marks above these

```
(LAMBDA (X N)
        (COND ((        N ≠)
              (NLISTP X))
              (T (AND (LISTP (SETQ X (NTH N̄ X̄)))
                      (NLISTP (CDR X)))))))
```

Figure 13. Annotated browsing of the old EQLENGTH

```
(LAMBDA (X N)
        (COND ((ILESSP N 0)
               NIL)
              ((      N)
               (NLISTP X))
              (T (AND (LISTP (SETQ X (NTH X̄ N̄)))
                      (NLISTP (CDR X)))))))
```

Figure 14. Annotated browsing of the new EQLENGTH

A-59

program elements are arrows denoting permutation. The direction of an arrow indicates in which direction one must look to find the corresponding element in the other version.

In addition to the visual annotation, a set of commands has been added to the File Browser for use when browsing an annotated file. The commands have to do with positioning the cursor in the file. One command searches forward from the current position to find the next annotation in the file; a parallel command searches backward to find the previous annotation. The third and final relevant command works only if the current position is at an annotation; in this case, it finds the counterpart in the other version of the file and sets the current position of that file to its annotation.

This last command is quite useful as it explicitly indicates which elements in the two versions have in fact been matched by the Heuristic File Comparator. Moreover, when numerous insertions, deletions, or long distance permutations cause corresponding elements to fall in radically different parts of the respective files, this command directs the user to the precise location of the matching element.

Integrating the File Browser and the Heuristic File Comparator has yielded a very useful tool for software development. The augmented File Browser provides just the right interface for the user to control inspection of the differences found between two versions of the same file. In future work, we hope to extend this capability to all list structured objects in the programming environment.

6.2.3 Code Presentation

One of the essential properties of computer programs that makes them so valuable is the ability to provide flexible behavior. This flexibility comes at a cost of program complexity, however, as flexibility is embodied in multiple execution paths. Several related paths are often aggregated within a single body of code, say a function or a procedure, and some branching control structure is used to decide which path to choose based on available data. Moreover, branching paths can be embedded within other branching paths. As a consequence, it can be very difficult to discern the path of execution through a given piece of code in different contexts.

Figure 15 is an example of such a piece of code. This code is parameterized by the two variables OBJECT and OPERATOR and is intended to return the name of the function which performs the prescribed operation for the given object type. For example, the PLUS operator for the INTEGER object is IPLUS. This code is implemented with the case statement SELECTQ and, in fact, has SELECTQ's embedded in each of the alternatives of the containing SELECTQ. The major SELECTQ branches on the object type and chooses a minor SELECTQ which is particular to that object type. The minor SELECTQ then distinguishes among operator types and locates the proper return value.

We have been working on a programming tool, which we call the Code Presenter, that is intended to make code such as this more comprehensible. The idea is to construct a hypothetical execution environment in which to view the code. The state of this environment will then restrict the possible paths through the code. If all of the state relevant to the code is defined, a unique path will be determined. If only a partial state is defined, the set of possible paths will simply be reduced.

```
Scrollable PP Window

(SELECTQ OBJECT
        (INTEGER (SELECTQ OPERATOR
                        (PLUS (QUOTE IPLUS))
                        (DIFFERENCE (QUOTE IDIFFERENCE))
                        (TIMES (QUOTE ITIMES))
                        (QUOTIENT (QUOTE IQUOTIENT))
                        (PRINT (QUOTE PRINT.INTEGER))
                        (ERROR OPERATOR "IS AN UNKNOWN OPERATOR FOR INTEGERS")))
        (REAL.NUMBER (SELECTQ OPERATOR
                        (PLUS (QUOTE FPLUS))
                        (DIFFERENCE (QUOTE FDIFFERENCE))
                        (TIMES (QUOTE FTIMES))
                        (QUOTIENT (QUOTE FQUOTIENT))
                        (PRINT (QUOTE PRINT.REAL.NUMBER))
                        (ERROR OPERATOR "IS AN UNKNOWN OPERATOR FOR REAL NUMBERS"))
        (LIST (SELECTQ OPERATOR
                        (FIRST (QUOTE CAR))
                        (SECOND (QUOTE CADR))
                        (THIRD (QUOTE CADDR))
                        (REST (QUOTE CDR))
                        (PRINT (QUOTE PRINT.LIST))
                        (ERROR OPERATOR "IS AN UNKNOWN OPERATOR FOR LISTS")))
        (ERROR OBJECT "IS NOT A KNOWN DATATYPE"))
```

Figure 15. Code segment parameterized by the variables OBJECT and OPERATOR

A-61

```
(SELECTQ OPERATOR
        (PLUS (QUOTE FPLUS))
        (DIFFERENCE (QUOTE FDIFFERENCE))
        (TIMES (QUOTE FTIMES))
        (QUOTIENT (QUOTE FQUOTIENT))
        (PRINT (QUOTE PRINT.REAL.NUMBER))
        (ERROR OPERATOR "IS AN UNKNOWN OPERATOR FOR REAL NUMBERS"))
```

Figure 16. Code segment with OBJECT bound to REAL.NUMBER and OPERATOR unbound

```
(QUOTE FTIMES)
```

Figure 17. Code segment with OBJECT bound to REAL.NUMBER and OPERATOR bound to
TIMES

```
(SELECTQ OBJECT
        (INTEGER (QUOTE ITIMES))
        (REAL.NUMBER (QUOTE FTIMES))
        (LIST (ERROR (quote TIMES)
                     "IS AN UNKNOWN OPERATOR FOR LISTS"))
        (ERROR OBJECT "IS NOT A KNOWN DATATYPE"))
```

Figure 18. Code segment with OBJECT unbound and OPERATOR bound to TIMES

A-62

In figure 15, we see the code segment presented in its entirety. This is the presentation one gets when both of the variables OBJECT and OPERATOR have no value. In figure 16, one sees the reduced code when the variable OBJECT has been bound to REAL.NUMBER. This minor SELECTQ is the only relevant code segment under this condition. Binding the variable OPERATOR to TIMES further restricts the code so that, as shown in figure 17, only the final result, FTIMES, is left. Figure 18 shows yet another interesting view of this code in the case that OPERATOR is bound to TIMES and OBJECT remains unbound. Here each of the minor SELECTQ's has been reduced to its respective case chosen by the value of OPERATOR and we see the major SELECTQ from this perspective.

The Code Presenter has three facets. The first is the maintenance of the hypothetical execution environment which forms the context for code presentation. The second is the ability to simulate the code so that execution paths can be determined according to the constraints of the execution environment. Finally, the code must be presented to the user to show the results of the simulation. We discuss each of the facets below.

There are two general cases in which a user might want to inspect code. The first is when he/she merely wants to examine the code under different conditions so as to provide insight on the structure of the code. The second is when the user is actually running some code and wants to inspect it in the current runtime context. The hypothetical environment and the simulation work together to support both cases. When the simulation needs the value of a variable, it first looks in the hypothetical environment to see if it has a value there. If not, it then looks in the real execution environment. This way the user can use the current context if he/she desires or can override it with a hypothetical case. He/she can even make a variable bound in the real environment appear unbound via the hypothetical environment. Functions are provided to set and unset individual variables as well as to initialize the entire hypothetical execution environment.

Simulation is used to determine which paths of code can potentially be executed within a partially or wholly specified environment. The reason simulation is used is to avoid any possible side effects the code might cause in the real environment were it to be directly executed. In addition, direct execution would not provide all of the information we desire.

Simulation is achieved by examination of the function calls which are composed to form the target code segment. The Code Presenter is endowed with knowledge about system functions so it knows which system functions it can simulate and how to do so. For calls to user functions, it obtains the relevant function bodies and continues the simulation. For functions it cannot simulate, it simply goes no farther. Simulation of computation also depends on the values of variables. When these are unavailable, once again the simulation must stop.

When simulation is complete and the information has been gathered about viable execution paths, there still remains the problem of presenting this information to the user. The most obvious and straightforward way and the one we pursued first is to present an edited version of the code which retains only the viable execution paths. This is the method shown in figures 16, 17, and 18.

An alternative way to present this information would be to show the entire code segment but highlight or distinguish the viable paths. For example, one might show

such paths in a different font which is larger or perhaps bold. The advantage of this approach is that it shows the contrast between paths which are viable and those which are not in context. We intend to investigate this presentation style in the coming months.

A preliminary version of the Code Presenter has been implemented and runs successfully on examples such as that shown in the figures. In future work, we would like to make it more robust by increasing its knowledge for simulation. We would also like to try other schemes for presenting the results. Our initial work suggests that this tool is a valuable one.

6.2.4 Graphical Debugging

Debugging any complex computer program has long been a difficult enterprise. This is due to the fact that most of a program's behavior has no direct visible consequences. The chain of events between the execution of an erroneous piece of code and overt evidence of program malfunction is often a long and complicated one. Consequently, the art of debugging is very much a case of detective work to infer the true culprit from available evidence.

Debugging techniques commonly involve modifying the target program in ways that produce more overt behavior. These modifications act as probes into the code so that more evidence can be gathered for the detective process. It is worth noting, however, that since computers can execute millions of instructions per second, one simply cannot make every action of the program manifest. To do so would produce an amount of information impossible to sift through, with most of it being useless. The trick is to make the right choice about what to show so that one can home in on the problem quickly and directly.

A common debugging technique is called tracing. The idea is to place a probe at a function call interface; that is, a function can be modified so that information can be presented to the programmer when the function is entered and when it is exited. Typically, together with the function name, arguments are printed on entry and return values are printed on exit. Several functions can be modified this way so that as the program is executed, one sees a (sparse) trail of the computation path.

Tracing is a valuable technique. however, the linear presentation of printed information it provides fails to carry with it the abstract structure of the program, which is an important context in which to interpret the information. Such a context is presented well by the Program Browser written for Interlisp-D and available in Interlisp Jericho.

The Program Browser constructs and visually depicts a graph of functions and their calling relationships. Figure 19 is an example for a program named HANOI which plays a visually animated version of the game Towers of Hanoi. The nodes of the graph are function names and the children of any node represent functions called by the parent node. Thus, in the example, HANOI is the top-level function and it calls, among others, OTHERPEG, CREATEDISKBM, and HANOI1. Under standard use, the graph is a tree, so that if a function is called by two different functions, it will appear as the child of each and hence will be represented by two nodes in the tree. Such nodes can be marked with surrounding rectangles to indicate multiple representation as is the case with the function OTHERPEG.
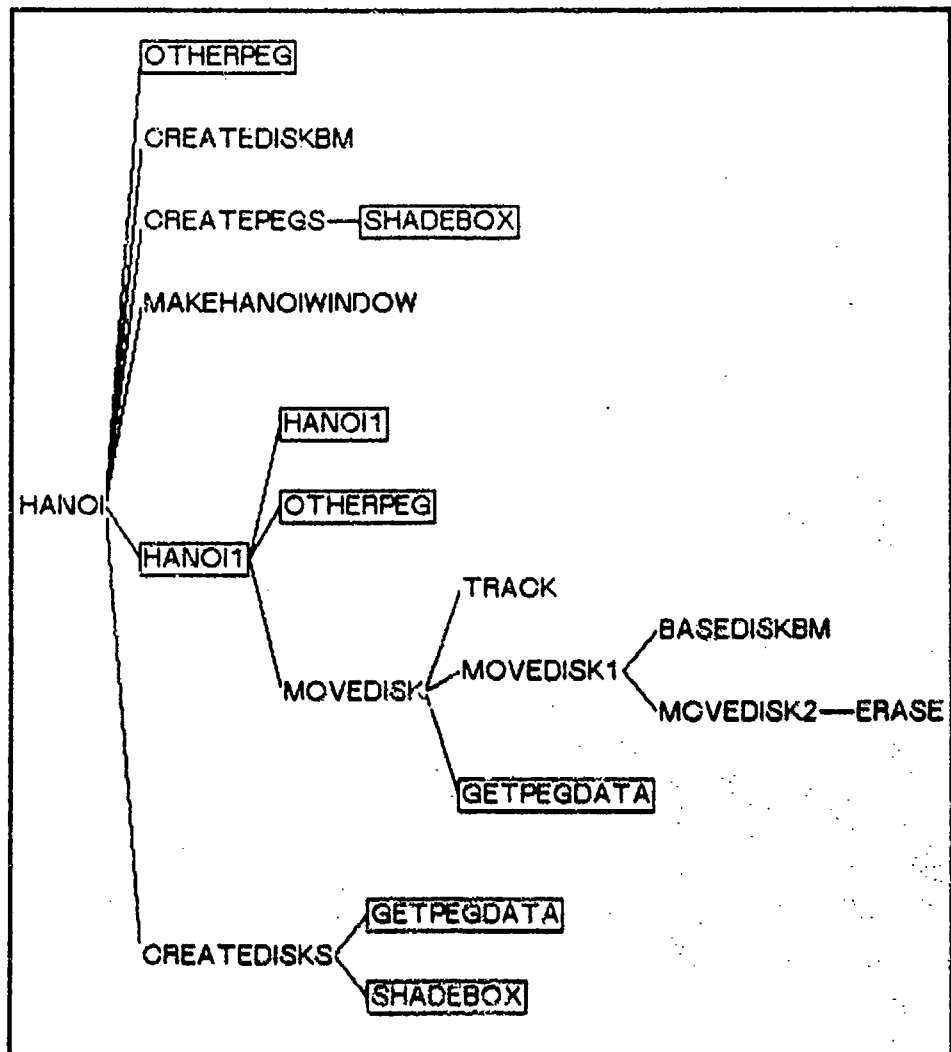
Figure 19. Program Browser for the program HANOI

In addition to its visual presentation of program structure, the graph nodes of the Program Browser are active. Using the pointing device, the user can select a node representing a function for three purposes.

o Printing, for which the source code of the selected function is printed in a window

o Description, which presents the results of an analysis of the selected function in a window; functions called and variables bound or used freely are examples of such results

o Editing, which invokes the editor on the selected function

We have augmented the Program Browser with a set of capabilities, most notably a tracing facility, so that it can now act as a user interface for program debugging. We call this tool the Graphical Debugger.

When the debugging capability is invoked, every function represented by a node in the Program Browser graph is modified to participate in a dynamic graphical trace. When a function is called, its node is highlighted to show execution is within that function. When it calls a subsidiary function, its highlighting remains but is diminished and the new function is highlighted. Thus, at any given instant of time, a partial branch of the tree is distinguished and, in fact, is a representation of the execution stack.

In Figure 20, one sees a snapshot of HANOI with the path HANOI, HANOI1, MOVEDISK, and MOVEDISK1 marked. MOVEDISK1 was the function being executed as a substep of MOVEDISK when the snapshot was taken. When the program is in progress, patterns of activity can be discerned as the highlighting moves around the graph. In this example, one immediately notices that after the program is initialized, all execution is localized in the subtree rooted at HANOI1 and, within that, predominantly in the subtree rooted at MOVEDISK.

The dynamic trace imposes only a moderate overhead on execution speed so that the highlighting moves around the graph rather rapidly which gives a nice global sense of program activity. However, the pace is too brisk to allow discrimination of intricate ordering relationships. As a consequence, we added the ability to induce a pause as each function is entered and exited. The duration of the pause can be varied dynamically so that activity can be slowed a lot or a little. This allows the user to quickly get to the interesting program component and then slow things down for detailed inspection. Indeed, the pause can be made into a stop so that the program can be single-stepped.

Another way to control execution for debugging purposes is to impose a stop or break when a particular function is entered. This way, the program can run at full speed up to the point of interest. The user can then single-step or run slowly as he/she desires. The Graphical Debugger provides an interface for establishing and identifying such breaks. The user can request a break and then point to a node and the represented function will thereafter stop each time it is entered. The user can similarly request to remove a single break. Alternatively, the user can request to remove all current breaks. Yet another command will highlight all the nodes currently representing broken functions.
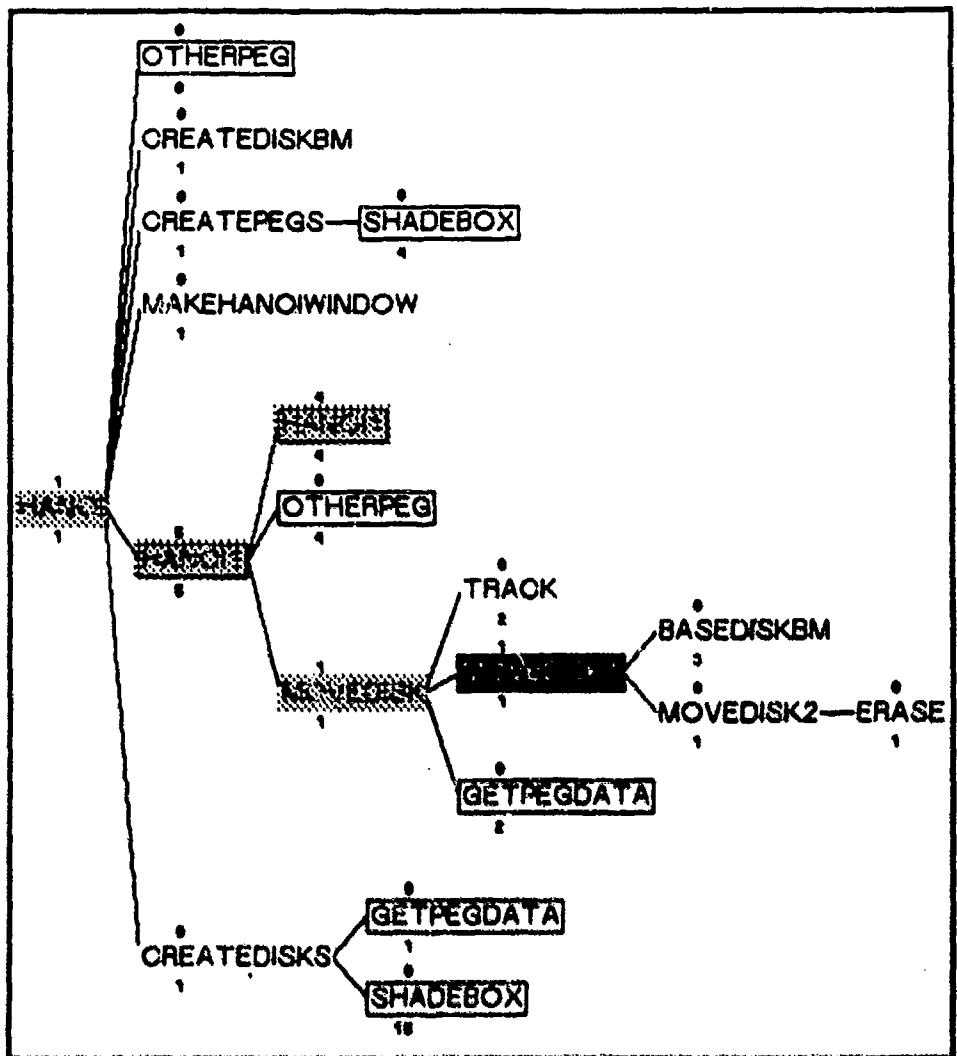
Figure 20. Graphical Debugger for the program HANOI

Above, we mentioned that trace facilities usually print arguments and return values as functions are entered and exited. The Graphical Debugger does not do this routinely. Instead, the user can ask for such information for the currently highlighted node. If that node gained control because it was just called, then its arguments are printed. If the node regained control because a function it called just returned, then the return value of the called function is presented. Perhaps we should provide the option of presenting such information for selected nodes automatically so that the user is not required to make a request every time.

Finally, the debugging capability also presents some other information displayed as numbers above and below the nodes in the graph as can be seen in Figure 20. The number above a node is a recursion count which denotes how many times the function has been invoked without yet returning. The node HANOI1 in the example has been called five such times. The number below a node counts how many times the function has been called at all. One sees that SHADEBOX has been called by CREATEDISKS in the bottom of the graph fifteen times.

The Graphical Debugger appears, from preliminary use, to be a convenient interface with which to develop programs. Seeing a dynamic trace provides an interesting and sometimes insightful perspective on program execution. The ability to control that execution via reference to functions through their representative nodes is efficient. More experience with this package will indicate how it might be improved and what long-term utility it can provide.

7. HERMES MAINTENANCE

During this reporting period the work on the Hermes mail handling program was routine maintenance.