M.I.T. Laboratory for Computer Science

Request for Comments No. 261
July 3, 1984

DSG Report Draft, July, 1983 -- June, 1984

by David D. Clark

Attached is a draft of the DSG Annual Report for 1983 - 1984. If you have suggestions for additions or improvements, please let me know, promptly.

DISTRIBUTED COMPUTER SYSTEMS GROUP

Principal Research Scientist

D.D. Clark, Group Leader

Research Staff

L.W. Allen
E.A. Martin
M.B. Greenwald

Graduate Students

R.W. Baldwin
W.C. Gramlich
K.R. Sollins
J.C. Gibson
P. Ng
L. Zhang

Undergraduate Students

D.A. Bridgham
T.H. Kim
M.B. Macaisa
H.J. Shinsato
G.D. Skinner
E.D. Crisostomo
J. Leschner
J.L. Romkey
E.H.Siegel
C.A. Warack

Support Staff

E.L. Felix

Visitors

A.J. Herbert

DISTRIBUTED COMPUTER SYSTEMS GROUP

1. INTRODUCTION

1.1. Introduction

The Distributed Computer Systems Group is a new group this year, formed from certain members of the Computer Systems and Communications Group and the Computer System Structures Group. The major project of this group has been the development of the Swift Operating System, but there are a number of other projects which are described in the sections to follow.

2. SWIFT

2.1. Project Summary

The Swift Operating System arose out of our earlier research in the implementation of network protocols. Recurring performance problems with protocol software led us to the conclusion that there was an underlying problem more general than specific design flaws in certain protocol suites. Our conclusion was that existing operating systems, such as UNIX, failed to provide the correct run-time support for highly interactive parallel software packages such as protocols. Swift is intended to demonstrate that proper support for this kind of software can be easily supplied.

The most novel aspect of Swift is the general structure provided for inter and intra process flow of control and information. Swift supports a programming style loosely built around two inter-related concepts, multiprocess modules and upcalls.

In traditional systems, the supervisor consists of a set of entry points which are invoked by the application program. That is, the application makes a subroutine call to a lower level, which performs some service for the application and then returns. In a network driven environment, most of the actions are initiated, not by the client from above, but by the network from below. Therefore, the most natural flow of control is not down from above but up from below. Most systems support this upward flow of control poorly, using either very inefficient interprocess message to achieve this upward flow, or a very efficient but unstructured interrupt. Our system permits subroutines to be arranged so that the natural flow of control can either be up from below or down from above. Permitting control to flow in a natural direction eliminates many unnecessary process schedulings within software such as network protocols, and has in some cases permitted a tenfold reduction in the bulk of the code, as well as a tenfold increase in its performance.

The upcall strategy eliminates process switching as a means of invoking a software module such as a protocol layer. Traditionally, a protocol layer would be organized as a separate process, but now it is organized as subroutines which live in a number of processes, each callable as appropriate from above or below. There must thus be a mechanism for these various subroutines to share state in such a way that the function of particular module is carried out. For this purpose, Swift uses the operarting system mechanism called a "monitor," a shared data object whose lock is managed by the system itself. Subroutines in different processes that collaborate with each other constitute a multiprocess module. Swift supports a programming style in which interprocess communication is never used as a way for one module to invoke another, but rather interprocess communication is only used within one module, through the mediation of the monitor locks.

Programs written using the philosophy of upcalls and multiprocess modules turn out, if written by sophisticated programmers, to be very simple, short and efficient. However, these tools, in the hands of tasteless programmers, have the possibility of creating the parallel programming equivalent of FORTRAN "spaghetti code." One of the concerns of our research is developing constraints on the programming style which lead to coherent and readable programs without severely impacting the efficiency and natural structure of the code.

There are two other features to Swift which, along with these new ideas for program structure, distinguish it from other operating systems. The first of these is the support which Swift supplies for real-time support, and the other is the support for management of its address space.

Swift is concerned with tasks such as network protocols, which involves scheduling a number of small computations to run in the 1-10 millisecond region. For this reason, the operating system that supports this task must have something in common with a real-time operating system, rather than a general purpose time-sharing system. Swift contains a task scheduler which assigns a priority to tasks based on the real-time deadline of the task, measured in milliseconds. This rather sophisticated scheduler is integrated into the monitor lock facility, so that a high-priority task encountering the lock set by a low-priority task can promote the low-priority task to run until such time as the lock is released.

The other important feature of Swift is its approach towards management of its address space. There are, historically, three ways toward dealing with the address space in an operating system. The first, typified by MULTICS, provides a multiple address space environment, with a very rich set of tools for sharing data between these address spaces in a controlled manner. This provided efficient interprocess communication, but required special hardware support which limited the portablity of the system. The second, typified by UNIX, provided a multiple address space

environment with limited tools for sharing between them. The elimination of sophisticated sharing mechanisms meant that there were no special hardware requirements for support of the system, so it could be easily ported onto new machines, but it meant that highly parallel computations were very difficult and inefficient to program on the system. For Swift, neither of these approaches appealed to us. We could not tolerate the inefficiency of a UNIX-like interprocess communication mechanism, and we were not interested in a system with strong requirements for specialized hardware support. Swift thus chose a third option for address space management, which is to put all the computations of the system into a single large address space. Clearly, this requires no special hardware support, and it certainly provides very efficient communication between different tasks. The major drawback of this approach, which is almost overwhelming, is that the broad sharing of a single address space means that program bugs cause corruption of arbitrary parts of storage, leading to crashes which are almost impossible to debug. We thus set ourselves a goal of building a single address space operating system with sufficient restraints that program development and execution would be stable and predictable, even though each task address space was nominally shared with all of the other tasks on the system.

The approach we took to this is to program Swift in a language which uses compile and run-time checking to detect erroneous references to memory and other similar bugs. Specifically, we have implemented Swift in the CLU Programming Language. CLU performs such tests as bounds checking for array references, and most important it prevents the use of an arbitrary bit-pattern as a pointer, so that references through arbitrary bit-patterns cannot corrupt unexpected locations in memory.

The most insidious bug which can arise in a system such as this is the use of a bad pointer, not because the pointer has been created incorrectly, but because the pointer points to a location in memory which has been de-allocated and reused. This can easily happen in a multi-task environment, if one task believes that an object is no longer needed, while another task continues to use that pointer. The only way to avoid this class of bug is to take de-allocation of storage away from the application and perform it in the system. This is the function called Garbage Collection. One of the challenges of the Swift system is to develop a garbage collector suitable for the operating system environment.

2.2. Project Status

A major effort effort over the last year has been the moving of the Swift system from the VAX on to a 68000 processor. This move was necessitated by several architectural limitations of the VAX, in particular the very inefficient I/O structure, by the lack of an all-points-addressable display for the VAX, and by the high cost of the

VAX, which prevented the deployment of the machine in suitable numbers. Swift is now running on a 68000-based machine which we have assembled out of commercially purchased cards. In retrospect, a great deal of group effort went into making usable a machine which was somewhat cheaper than would have been ideal: however, Swift is now running and we are proceeding with a number of improvements which were not possible within the VAX version of the system.

Considerable effort has been invested in the design of a suitable garbage collector for Swift. Ideally, our garbage collector would have the following characteristics. First, it is incremental, collecting garbage a little at a time while the system runs, rather than stopping the system while all of memory is examined. Second, it should not move objects around as a part of collecting garbage, because I/O devices may have pointers into memory, which are difficult to find and change arbitrarilly. Third, it should not require a large quantity of additional memory to use as the temporary storage area for the garbage collection process.

A number of very creative ideas were proposed for the Swift system, taking into account that the system is intended to run on a desktop workstation or similar node of a networked distributed system. If we truly believed that network communication was extremely rapid, then an obvious way to garbage collect one's environment is to make a snapshot of it and send the snapshot across the net to a cleaning machine which would send back a laundered version of the address space. However, it seems difficult to get the necessary throughput across the net. Certain garbage collection strategies, in particular the one proposed by Dijkstra, seem directly relevant to what we are doing and have been implemented for trial later this year. Perhaps the most novel garbage collector we have proposed is the Probablistic Parallel Garbage Collector (PPGC). This superficially silly idea involves allowing the garbage collector to run in parallel with the other tasks, without the traditional interlocks. This raises the possiblity that, under certain very anomalous circumstances, the garbage collector may declare as garbage something that is not. However, there are ways that this event can be tested after-the-fact, which means that the investment of additional background cycles from the machine can reduce to an acceptable level the probability that we have made an error. Unfortunately, we cannot come up with any analytical model that tells us exactly what the probability of failure is. Thus, we are experimenting with PPGC, under a variety of program loads, to determine under what circumstances errors may arise. It has been difficult to assess the probability so far, since we have not yet had a single garbage collection error due to the probabilistic assumption.

CLU, as it is used as an application programming language at M.I.T., does not have any dynamic linking capability. That is, one compiles all the programs that are to be part of the current run, then one statically links them, and then one brings this static load-image into execution. Swift requires a dynamic linking capability, and over the

last year this has been designed and implemented. We are now testing a dynamic loader which will bring sub-systems into a running CLU environment and dynamically resolve all of the cross-module references.

A more difficult problem is unlinking the module and removing it from the environment after it is needed. It could be argued that this step is unnecessary. However, this will cause the size of the link-image to grow continuously, which means that the memory management algorithm must be more sophisticated than if modules can be thrown out when they are no longer needed. Since we do not wish to make sophisticated assumptions about our virtual memory (since we want the system to be portable on to a wide variety of hardware) and since virtual memory techniques are fairly well understood, we are concentrating our energy on exploring the higher- level question of whether or not programs can be unlinked and removed from the address space of the system, either to suspend them while they are running or to remove them when they are no longer needed. Preliminary design for this approach is now completed.

Most operating systems provide a file system for their users and Swift is no exception. However, we were uninterested in writing the necessary device drivers to permit Swift to support disks. Instead, as part of Swift, we have developed a remote file system protocol, which permits Swift to utilize file systems stored on other machines. We have implemented a simple server for this protocol, and the implementation of the Swift interface is almost complete.

We have implemented a number of test applications on Swift, in order to detemine system performance and to stress the garbage collector. Perhaps the largest is the text editor TED, which is a very popular CLU program on both TOPS 20 and UNIX. As soon as the file system is working, we will be able to run a fully usable version of TED on Swift, which will give us a considerable test of the system's performance.

One of the most interesting applications has always been network protocols themselves, and we are currently rewriting our earlier implementation of the protocol package for Swift, in order to improve its performance, and to get more experience with the proper structure of such programs inside Swift.

2.3. Future Directions

The major implementation effort on Swift will probably finish within the next six months. At that point there will be a major project review to determine what future direction the project will take. There are two important questions which we wish to answer before the project terminates. The first of these is whether we can understand how to structure programs built around the concepts of upcalls and multiprocess modules. These concepts give the programmer great freedom, and it

has been noticed in the past that great freedom sometimes leads to very poor programming style. We thus identify a conflict between our desire for the efficiency which comes from the these new tools, and the desire to have contraints that lead to good style. We feel that Swift has taught us some fresh insights about the construction of multiprocess programs, and we are anxious to understand this in as general a manner as possible. Second, we are anxious to learn how effectively we can achieve our goal of supporting reliably a single address space operating system. In order to this we must experiment further with garbage collection and with linking and un-linking.

In order to achieve these goals, we feel that it may be necessary to implement additional application programs which utilize the features of Swift. However, the hardware base on which we run is sufficiently unstable that large-scale software development is not tractable. We must, therefore, make a decision as to whether we will move Swift onto yet a third hardware base.

2.4. Related Activities

There are a number of small activities which, although not strictly a part of the Swift Project, are closely associated with it within our group. These include our support of UNIX, which is used as the program development environment for Swift, development of a tasking package for the IBM PC which supports many of the same programming conventions of Swift (most particularly upcalls), and the support of the Remote Virtual Disk Protocol.

Remote Virtual Disk (RVD) is a protocol which lets one machine attach to a file on a server machine and to use that file as if it were a disk. Our group initially developed RVD to expand the disk space on our VAXes, and the tool has now become a widely used facility within the VAX-UNIX community of LCS. However, the server which we initially implemented, which was not really intended to provide serious operational support, needs rewriting in order to be sufficiently robust, and the Computational Resources Group has undertaken the task of producing and running a better server for RVD. User programs for RVD exist for UNIX 4.2, and for the IBM PC.

3. DISTRIBUTED ARCHITECTURES FOR MAIL

For many users, computer mail has been the most important application of computer networks. The software for distribution and forwarding of mail is very well developed at this point, but the software for displaying, generating and archiving mail is still based on the assumption of a centralized time-sharing machine. The goal of this project is to develop a user interface to a mail system which is suitable for a personal computer model of distributing computing.

In the centralized view of mail, each user is served by a particular machine on which is located the user mailbox, as well as the necessary software to manipulate this mailbox. The assumption is that the user directly logs in to the machine on which the mailbox is located. The machine containing the mailbox uses some standard mail forwarding protocol to communicate with other mailbox machines located around the network, in order that mail is properly forwarded. For this reason, the mailbox machine must be available as a server on the network essentially all the time.

Since the mailbox machine must operate like a server, it is not appropriate for a personal computer to be the mailbox machine. A personal computer may be powered-off much of the time, and even if it is physically powered-on and connected to a network, may not be able to run server code as a background job.

We have developed a new architecture for mail software, to cope with the characteristics of a personal computer. Our design divides the traditional functions of a mail processing node into two parts, the management of the mailbox and the execution of the user interface software. These functions are implemented on different machines; the mailbox is stored on a centralized server called a repository, and the user interface software runs on a personal computer.

The goal of this research is twofold. First, this project is attempting to produce software that can be used in practice. Several of our existing mail nodes are heavily overloaded, so the ability to read mail on a personal computer would simultaneously remove a burden from our centralized time-sharing systems, as well as providing a mail processing environment which is more responsive and interactive. More importantly however, this research has a goal of exploring how traditional applications should be retructured in the context of a distributed computing environment. Several interesting problems arise, which we cannot yet solve in general, but which we can explore using mail as a particular example.

The first problem is that the personal computer may not have continuous access to the data stored in the repository. In fact, the personal computer may be diconnnected from the net much of the time. In particular, we would like to permit the user to send and receive mail at a time when a network connection is not open. This will permit a portable computer to be used as an interface to the mail system. This means that a temporary copy of the user's mailbox must be created in the personal computer, matching as closely as possible the master copy which is stored in the repository. We thus have multiple copies of the database describing the mailbox, which are almost always partitioned, but only occasionally able to talk to each other. Updates can occur to either copy, and the software must do its best to keep all of the versions consistant.

The second problem is that the user may wish to interact with the mailsystem from

a number of personal computers, and he would like to see a consistant view of his mailbox, no matter which personal computer is acting as a frontend. This means that in addition to the master copy stored in the repository, there may be several rather than one auxiliary copies, each of which has a slightly different version of the mailbox in it. This, plus an additional requirement that the system must recover whenever a personal computer crashes and looses its copy of the mailbox, makes very difficult the problem of keeping the user's view of the mailbox consistant.

During the last year, we have designed the mailbox repository, the user interface, and the protocol which hooks them together. The protocol contains some rather sophisticated error recovery mechanisms, so that the connection can be disrupted at an arbitrary point without causing the various copies of the mailbox to diverge in an irreconcilable manner. We have a prototype implementation of the repository, and over the next six months we intend to demonstrate a running repository and a user interface program running on an IBM personal computer.

4. NETWORK ROUTING AND RESOURCE CONTROL

Elizabeth Martin continued to study dynamic routing problems in an internet. We have been participating in the development of the Exterior Gateway Protocol, which will be used to pass routing information between gateways in the DARPA internet, and have proposed some schemes for dynamic routing within the MIT internet, a subsection of the DARPA internet.

4.1. Exterior Gateway Protocol

The Exterior Gateway Protocol partitions groups of gateways in the DARPA internet into Autonomous Systems of gateways. The gateways in an Autonomous System(AS) are programmed, maintained, and administered by one organization. Gateways in different Autonomous Systems exchange routing information using EGP. EGP consolidates the amount of routing information that is exchanged and provides a more controlled method of passing information between gateways who may or may not trust eachother. Gateways within an AS use their own conventions for passing routing and up/down information among themselves; these gateways are free to experiment with different routing algorithms.

Defining rules for the internet routing topology and for the dispersal of routing information that prevent routing loops has proved to be very difficult. Consequently, the early version of EGP which is expected to become a DARPA standard this summer is very restrictive.

We have participated in specifying EGP and in studying the routing issues this protocol is attempting to address. We have implemented EGP for our C gateway.

4.2. Interior Gateway Protocol

The MITnet consists of about 45 interconnected LANs (called subnets), and is expected to grow to about 200 LANS by 1990. It is becoming increasingly inconvenient to manually change the subnet routing tables in all the MIT gateways every time a new subnet and gateway is added to the MITnet. Therefore, we have specified a subnet routing protocol (our Interior Gateway Protocol) to be used to tell gateways within the MITnet 1) which gateway to use to get to which exterior nets outside the MIT network and 2) which neighbor gateway to use to get to the various MIT subnets.

This IGP should serve the MITnet's needs for the next five years or so at which point the amount of routing information that must be processed may become too cumbersome.

4.3. Internet Congestion Control

Lixia Zhang began a study of network resource allocation techinques suitable for the DARPA Internet. The Internet currently has a simple technique for resource allocation, called "Source Quench."

Simple simulations have shown that this technique is not effective, and this work has produced an alternative which seems considerably more workable. Simulation of this new technique is now being performed.

5. DISTRIBUTED NAME MANAGEMENT

Karen Sollins continued work on her doctoral thesis, "Distributed Name Management." This research is being supervised by Prof. David P. Reed. Based on identification of issues underlying name management in a distributed computing environment, a naming framework has been developed and applied in two domains, electronic mail and software management.

5.1. Introduction

Names form the basis of all communication, of all verbal expression among people. They also form the basis of much of private cataloging and record keeping. Without some agreement on or understanding of names, most interpersonal, and even much personal record keeping would be hampered. If communication is with or through a computer, all communication would cease without a common understanding of names and their meanings.

5.2. The Problem

If we look at the history of naming in computer systems, we see a progression toward making names more useful to and usable by humans. In the earliest systems, every nameable entity had a globally unique name. In the last resort, these names were physical addresses. Early on, in order to help organize and segment the name spaces used by humans in communicating with computers, hierarchically organized directory systems and user naming facilities were developed. These required humans to remember only small subsets of the names and provided some hints for large parts of the names. But, there was generally little flexibility in generating such names. Users were assigned directories; names within them had limits on the number of characters or form. An improvement on this situation was to allow the user to define private nicknames. Linking in Multics is an example of this. Nicknaming improved the situation for the individual, but still did not address problems of sharing or cooperatively defining names using computer systems as the medium and the source of objects to be named.

One of the purposes of a naming faility is to provide humans with names. In addition, humans provide a good paradigm for studying cooperative name management among a group of relatively autonomous participants. Therefore, we have identified eight observations about human naming to guide in the determination of the framework.

1) **Communication:** *Names are the basis for communication. Therefore sets of names used by individuals should be sharable, reflecting common interests and communication patterns.*

2) **Multiplicity of names:**

   - *Different people use the same name for different things.*
   - *Different people use different names for the same thing.*
   - *A single user uses different names for the same thing.*
   - *A single user uses the same name for different things in different situations or at different times.*

3) **Locality of names:** *A person uses sets of names to reflect his or her focus of interest. A user also may use two or more sets of names to reflect a focus between or including several focii of interest.*

4) **Individuality:** *Each participant carries a personal environment that he or she can bring to bear at any time. For example, part of that personal environment includes the names defined in conversations other than the*

one currently in progress. Various parts of the environment may take on more or less importance to the individual in different situations. The individual manages this personal environment for each topic privately by specifying a mutable partial ordering on the parts of the environment, and using them as sources of candidate names for the topic at hand.

5) **Flexibility of name usage:** *Humans use several sorts of names. For example, names are often descriptions. People use both full and partial descriptions. Humans also use generic names to label classes of objects. These generic names may be labels or descriptions. In fact, humans often use combinations of generic names and descriptive names.*

6) **Manifest meaning of names:** *The meaning of a word used by humans is constrained by human language in a way that is understood by other humans as well.*

7) **Usability of names:** *Humans are able rapidly to define or redefine names and shift contexts on the basis of conversational cues. They also have mechanisms for disambiguating names, such as querying the source of a name for further information.*

8) **Unification:** *Humans apply a set of naming conventions uniformly to all types of things. This is in contrast with more automated situations in which the particular method of indicating an object is based on the nature of the object itself. For example, in many computer systems, user names must be in one restrictive form, file names, another, process identifiers a third, etc.*

5.3. The Framework

The framework that this research proposes consists of a new type of object and the facilities needed to support and manipulate it as a source and repository for cooperatively managed names. An individual accesses the names through an **aggregate**, the user's private view of a name space shared with other users. An aggregate consists of a shared **context** and a partially ordered set of contexts from which the individual may wish to draw names and objects. A context is a mapping from names to named objects. Within a context there are no restrictions that names be unique; i.e. a name can be applied to more than one object within the same context. Indirection is also provided, allowing a name to be resolved into another name. Operations on aggregates include the ability to create and delete aggregates, create, delete, examine and modify entries, manage the partially ordered set of background contexts (which involves being able to name contexts), and move from

one aggregate to another (which involves being able to name an aggregate from within another aggregate).

5.4. Application of the Framework

The framework is applied in the areas of electronic mail and software management. To date the first has been implemented; the second will consist of a paper design. In the mail project, contexts and aggregates are used to manage the names of mail recipients for both incoming and outgoing mail. The mail itself is the vehicle for transfer of information, including both the user community's choice of names for individuals and the network mailbox identifiers for the same individuals. Participants in a "conversation" indicate the aggregate name in an additional field in the message. When a message is sent or received, a filtering process translates the names in the various mail header fields using the specified aggregate. A subprocess running under the user's mail reading program has access to all the user's aggregates and provides the translation service. The implementation is on a VAX running 4.2 Berkeley Unix and is written in a combination of CLU and Mlisp, an extension language of Gosling's Emacs.

Among the issues brought to light by the electronic mail implementation, the question of name addition and deletion from the aggregate is especially interesting and novel. In human name management, when a name is proposed as part of a cooperative effort, the name passes through several stages. It is proposed initially. As it is used several times, it becomes accepted. In addition, it may experience modification, typically simplification, during this process. If it falls into disuse, it may need further explanation to refresh other participants' memories. After a period of disuse it may finally in time become completely obscured. The issues raised include: the possible states of a name, the criteria for accepting and obscuring names, and methods of keeping local copies of a shared context in synchrony.

5.5. Contributions

The contributions of this work lie in three areas. First the research extends our understanding of naming requirements by recognizing three factors that enhance the utility of names: sharing, uniformity, and the uniqueness of the individual users. The second contribution is the approach taken to solving the problem. Sollins proposes non-hierarchical, small, shared namespaces coopertively managed with the provision of additional, privately defined sources of candidate names to complement the shared namespaces. Finally, with the completion of the thesis Sollins will have categorized issues in naming by whether they are application specific or universal naming issues.

6. CHECKPOINT DEBUGGING

Wayne Gramlich continued work on his thesis in the area of debugging distributed systems.

The specific debugging technique that is being investigated is called checkpoint debugging.

The basic idea is to regularly take an atomic snap-shot of the process state and then record all subsequent process input until the next atomic snap-shot. When a bug is encountered, the previous snap-shot and all of its process input is retained for subsequent analysis. Debugging is performed by reloading the snap-shot and replaying the process input.

Since computers are deterministic finite state machines, the sequence of events leading up to the occurance of the bug can be recreated as many times as necessary. A conventional interactive debugger can be used to help find the bug when the checkpoint is being replayed. For a distributed computation, all the processes in the computation must be regularly checkpointed. When a bug is encountered in any of the processes, the checkpoints for all processes must be retained. Checkpoint debugging can also be effectively used to help debug real-time systems. Checkpoint debugging is also useful for debugging high availability programs (such as printer servers) where the system maintainer can not be available for debugging 24-hours a day.

References

Publications

1. Saltzer, J.H., Reed, D.P., and Clark, D.D., "End-To-End Arguments in System Design," to appear in ACM Transactions on Computer Systems, November, 1984.

2. Greenwald, M.B., "Remote Virtual Disk Protocol Secification," MIT LCS Technical Memorandum (in preparation -- expected publication date 1984)

Theses Completed

1. Gobioff, B., "An Investigation of Development Methodologies for Communications Software," M.S. thesis, MIT, Sloan School of Management, Cambridge, MA, May, 1984.

2. Kim, T.H., "A Distributed Mail System Repository," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1984.

3. Krajewski, R.P., "Required Capabilities for a File Access Protocol," S.B. thesis, MIT, Department of Electrical Engineering, Cambridge, MA, May, 1984.

4. Shinsato, H.J., "A CLU Interface for a Bit-Mapped Display," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1984.

5. Skinner, G.D., "An Implementation of an ARPANET FTP Server for UNIX," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1984.

6. Spurlock, J., "A Comparative Study of Distributed File Systems," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1984.

Theses in Progress

1. Siegel, E.H., "Dynamic Linking in a Type Safe Environment," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, September, 1984.

2. Gramlich, W.C., "Checkpoint Debugging," Ph.D. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, September, 1984.

3. Romkey, J.L., "Reliable Datagram Multicast on the Internet," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, December, 1984.

4. Sollins, K.R., "Distributed Name Management," Ph.D. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, August, 1984.

Talks

1. Clark, D.D., "Remote Virtual Disk Protocol," Internet Research Group, January, 1984.

2. Clark, D.D., "A Case Study: The Campus Network Plan for the Massachusetts Institute of Technology," ACIS, IBM, Rockville, MD, January, 1984, March, 1984.

3. Clark, D.D., "Overview of Research at Massachusetts Institute of Technology, Laboratory for Computer Science," Digital Equipment Corporation, Hudson, MA, March, 1984.

4. Clark, D.D., "The Reality of the Newtwork Protocol Jungle," MIT Industrial Liaison Program, Cambridge, MA, April, 1984.

5. Greenwald, M.B., "Swift: An Operating system for a Personal Computer," Ninth ACM Symposium on Operating System Principles, Bretton Woods, N.H., October, 1983.

6. Greenwald, M.B., "Accessing Secondary Storage Across a Data Network," Digital Equipment Corporation, Littleton, MA, June, 1984.

7. Martin, E.A., "MIT Gateway Projects: Campus Network/Project Athena and Dynamic Routing," Gateway Special Interest Group Meeting, USC Information Sciences Institute, Marina Del Rey, CA, February, 1984.

8. Sollins, K.R., "Distributed Name Management," Ninth ACM Symposium on Operating System Principles," Bretton Woods, N.H., October, 1983.

Conference Participation

1. Clark, D.D., Panel Session, ACM Sigcomm '84, Communications Architectures and Protocols, Montreal, Quebec, Canada, June, 1984.

Committees

1. Clark, D.D., Chairman, DARPA Internet Configuration Central Board and Internet Research Group.

2. Clark, D.D., Treasurer, Panel Session and Program Committee Member, Ninth ACM Symposium on Operating System Principles, Bretton Woods, N.H., October, 1983.