M.I.T. Laboratory for Computer Science

Request for Comments No. 263
September 24, 1984

CS/DSG Report, July 1982 -- June 1983

by D. D. Clark and J. H. Saltzer

COMPUTER SYSTEMS GROUPS JOINT REPORT

1. INTRODUCTION

1.1. Introduction

The Computer System Structures Group and the Computer Systems and Communications Group have been working jointly on a series of related projects that are described in this joint report. The projects cover quite a wide territory:

1) The largest effort was the development of Swift, an operating system kernel with several unique features: multiple tasks in one address space with compile-time protection, heap allocation with garbage collection, and upcall-downcall organization. Swift captures in a single design many ideas developed in the last ten years on how to integrate network communications and display management with an operating system kernel.

2) SWALLOW is a unique remote file storage system that uses an append-only user interface to provide atomicity of file updates.

3) A thesis explores whether names used in computer systems can more closely resemble the way names are used by people.

4) The interconnection of networks belonging to different organizations is the subject of a new research direction that explores both the technical and policy issues that are raised.

5) The Community Information System is a new, experimental approach to dissemination of information using radio broadcast of large volumes of data and selective filters implemented by intelligent receivers.

6) A Remote Virtual Disk protocol was designed, implemented, and placed in service as part of a project to explore models of personal computing.

7) The experience of several years in designing network protocols with modularity violations was captured in a thesis on "soft layering."

8) The IBM personal computer was turned into a full scale network host attached to the Ethernet, with file transfer and remote login facilities.

9) Work continued on Internet protocol and gateway implementation,

including especially an exterior gateway protocol that permits isolation of the internals of one organization network from another.

10) A ring network monitoring station was completed. Statistics can now be gathered concerning the relative effectiveness of token rings and Ethernets.

11) An online directory assistance system, DIRSYS, provides a unique interface to a telephone and mailbox directory of 20,000 names.

These eleven projects, and many sub-projects, are individually described in the sections that follow.

2. SWIFT

2.1. Milestones

The past year has seen significant progress in the Swift effort. The first eight months, roughly, were devoted to planning the implementation effort, including design of the tasking and memory management systems. Implementation started at the end of January of this year. A stand-alone system with rudimentary memory management and a preliminary implementation of the tasking system was running within a month. This implementation also included a simple driver for the console terminal and support for timers. The next month was devoted to the development of network code, including a driver for the 10 Mbit/sec token ring network developed by our group[1] and an implementation of the DoD Internet protocol. By mid-Aprilthe system was regularly sending and receiving Internet packets, and detailed performance measurements were under way. Also in April, a Remote Debugging Protocol was designed and implemented.

The detailed performance measurements carried out during April pointed out many areas where performance could be improved, particularly in the tasking system and the I/O device management. As a result, large portions of the system are being reimplemented at this time.

[1] Being sold commercially by Proteon Associates as proNET.

2.2. Lessons

2.2.1 Programming Language

We have gone against traditional operating system lore and implemented Swift in CLU -- a high-level, strongly typed, object oriented language. This is not a radical departure; after all, Multics was written in PL/1, UNIX in C, and Pilot in Mesa. The argument against implementing an operating system in a high-level language is generally efficiency: there is a general belief that high-level languages cannot generate appropriately efficient code for the internals of an operating system. The arguments in favor of implementing an operating system in a high-level language generally include portability and ease of writing, debugging, and maintaining code -- the standard arguments for high-level languages, in general, over assembly language. Operating Systems are written in a high-level language because the designers believe that the loss in efficiency is outweighed by the benefits of a high-level language.

There are additional arguments for implementing Swift in CLU beyond these traditional arguments. The CLU compiler enforces type safety, provides storage management, and provides and enforces useful data abstraction mechanisms.

Enforced type safety allows us to eliminate the kernel/user distinction, and to have many processes peacefully coexist in a single address space. We can trade off compile-time checking for the run-time overhead of protected areas of memory or code. This allows us to use the full features of the system even deep within the bowels of device drivers, historically a difficult place to program and debug code.

Storage management by a garbage collector allows us to freely use variable length data objects inside the kernel. A large class of operating system problems are related to allocating and freeing data objects and dangling references. We think that the system overhead of a garbage collector is a reasonable price to pay to avoid these problems.

One reason that we chose CLU is that we believed it would be easy to modify the compiler to support Swift. Though we haven't done this yet, our experience with CLU so far has been very positive.

2.2.2 Advantages of CLU

We were able to bring up a first system within a month of coding. Swift has been through one major rewrite already and currently consists of 4600 lines of CLU code, 1600 lines of machine language(ASM) code, and 600 lines of ASM code for the

remote debugger, RDB[2].

We found the same advantages implementing Swift in CLU that application programmers have found: increased productivity, the compiler found a good number of bugs, we were able to integrate different people's code easily, and strong typing did help correct bugs.

Swift, like all operating systems, had its share of bugs, but most of them were in the ASM code. The CLU compiler detected most errors in the CLU code. After the ASM code stabilized, almost all the bugs that were not caught by the compiler were conceptual problems with the design of Swift, and not careless errors. This was partly because of careful coding on our part, and good fortune, but it has convinced many of us that the CLU compiler is well worth the cost.

2.2.3 Troubles with CLU

A good portion (1600 lines of code) of the operating system is written in ASM, the assembly language interface of the CLU developement system. Much of this was written in ASM for efficiency, not because it would have been impossible to write in CLU. However, most of the cost of these operations is the subroutine call necessary to invoke them. A good example of this is the word cluster -- a data abstraction we devised to deal with logical operations on 16 bit quantities. The overhead of a CLU procedure call is on the order of 20 microseconds, so to do a few simple operations on a word can take as long as 100 microseconds. If the optimizer performed in-line optimization on words, this problem would disappear.

Adding these optimizations to the compiler does not seem very difficult, and we plan to make these additions shortly.

Some characteristics of CLU plagued us throughout. A recurring problem is that of closely coupled abstractions: two data abstractions that together support an invariant. It would be useful if there were some way supported by the compiler to package the two together so that the representation of each was available to the other, but to no one else.

2

Lines		Code	Comments	Blanks
	CLU	4656	1166	1587
System	ASM	1611	830	408
	Equ	1421		
RDB	ASM	633	222	207

CLU is a high-level language, and the details of its implementation should be of no concern to the programmer. Yet, when writing an operating system, a certain amount of awareness is necessary. There are times, admittedly few, when allocations are forbidden. It is necessary to know when CLU is allocating objects from the heap. There are some expensive mechanisms under the covers that are not always obviously expensive -- programmers writing the internals of Swift must understand these.

A serious inconvenience throughout Swift was the problem of dealing with 32 bit quantities. CLU reserves a bit per longword to determine whether the longword is a reference or an integer. In application programs this rarely matters -- inside the operating system it can cause no end of trouble. There are many objects that are most easily represented by 32 bit quantities: virtual addresses, page table entries, and so on. We spent a considerable amount of effort trying to find schemes that would allow us to fit 32 bit integers into CLU, to no avail. The strategy that we adopted was to allocate objects in the heap to hold the 32 bit quantity, or store the 32 bit quantity in two integers. In cases where performance or space was a serious issue we resorted to ASM.

2.2.4 Deadline Scheduling

In our experience, computer operating systems often need to respond to some events in "real-time;" that is, with a guaranteed maximum latency. This is true even of systems not explicitly designed as real-time systems; it is especially important when network support is required. Thus one of the design requirements we have identified for Swift is that it be able to respond to events in real-time.

There are a number of problems involved in designing a system, especially a general-purpose system, with real-time capabilities. First, there is the issue of preemption. Maintaining a guaranteed maximum latency for response to events requires that activities in the system must be preemptible. Preemptive scheduling, of course, requires synchronization mechanisms to coordinate shared access to resources; these synchronization mechanisms may interact badly with the scheduling system employed, as will be discussed below.

In any uniprocessor multi-tasking system, the processor is a scarce shared resource, and hence its usage must be scheduled. To insure the desired low real-time latency in responding to events, most real-time systems define "priority" schemes, in which activities are ranked by order of importance and the most-important (highest-priority) activity is given the processor. The problem with such schemes is that the priorities have a global significance. The priority of a particular activity cannot be meaningfully assigned without knowing the priorities of all the competing activities in the system. In a general-purpose system like Swift, in which tasks are dynamically created and in which new programs may be run at any time,

this complete knowledge is impossible and hence traditional priority schemes are not suitable.

An alternative approach to processor scheduling may be motivated by going back to our original definition of "real-time response;" namely, response with a guaranteed maximum latency. This definition suggests a natural way to schedule the processor: each activity specifies to the scheduler its maximum allowable latency; from the latencies a *deadline* for each activity is computed, and the activity with the earliest deadline is run. This approach meets the modularity goals outlined above: any activity can be designed and specified without regard to which other activities may be competing. Moreover, it has the advantage that deadlines are a very natural concept for programmers to grasp, unlike priorities (which are meaningless numbers in and of themselves).

A problem often noticed with deadline scheduling schemes lies in the specification of the deadline to be met. Most programs are composed of a variety of independently-designed modules, and it would be desirable to allow each independent module to define its own scheduling behavior and deadline. To this end we have defined the notion of a "scheduler region." Each scheduler region may independently define the deadline for its own completion. The regions may be nested, subject to the constraint that the deadlines in nested regions must be monotonically increasing (this constraint is enforced by the software). Regions both aid in the modular development of software and assist in solving the monitor interaction problem, as described in the next section.

2.2.5 Interactions of Deadlines and Monitors

As mentioned above, the use of a preemptive scheduling system requires the introduction of synchronization mechanisms to coordinate access to shared data; we chose the *monitor* mechanism, which integrates nicely with CLU's clusters. Other researchers have noted that such synchronization techniques can interact badly with priority scheduling systems. For example, suppose that a high-priority task has to wait to enter a monitor held by a preempted lower-priority task. If there is another runnable task with an intermediate priority, it will be run next, and will effectively delay the execution of the high-priority task.

Evidently what is needed is a way to temporarily *promote* the task holding the monitor until it can get out of the higher-priority task's way. This is particularly clean in the case of a deadline scheduler: the deadline of the higher-priority task is *propagated* to the task holding the monitor until the monitor is released. This is implemented by causing the low-priority task to enter an "implicit scheduling region" in which it will remain until it leaves the monitor in question; its deadline while it is in the implicit region will be equal to the deadline of the high-priority task waiting for the monitor.

This simple form of deadline propagation does not suffice, for it is possible that the low-priority task which holds the desired monitor is itself waiting to enter a second monitor which is presently held by a third task; and so forth. We need to propagate the high-priority task's deadline through the entire chain of waiting tasks. We can do this in a particularly simple and clever way: after promoting the low-priority task to the high-priority task's deadline, we simply wake up the low-priority task. When the low-priority task is awakened, it will again try to enter the monitor for which it is waiting; deadline propagation will again be performed, and the process will recur until a runnable task is reached.

In practice, of course, we do not expect long chains of deadline propagations to occur. It is relatively rare for a task to go blocked with a monitor locked or to attempt to enter a nested monitor, so most of the time a deadline propagation is required, the task being promoted will be runnable. In this case the overhead required by deadline propagation is minimal, and the mechanism cheaply and efficiently solves the monitor interaction problem.

2.2.6 High resolution hardware clocks are essential

We have relearned the lesson that many people have learned over the years: a high resolution hardware clock is essential.

Swift has no interrupt handlers. When an interrupt goes off, a task is scheduled to handle it. We expect to be able to handle interrupts from devices that have a maximum latency of 100 microseconds. This requires our scheduler to be able to handle deadlines that are specified in microseconds. We also need microsecond resolution for metering the code.

The current implementation of Swift on the VAX tries to take advantage of the hardware interval timer provided.

There are two problems with the VAX interval timer for Swift. The first is the overhead associated with updating the software clock, due both to the VAX and to Swift. The second is that the clock is jointly updated: some parts of the clock are updated by hardware, and some by software, which causes serious interlocking problems. For this reason, we would strongly prefer a better clock supported in hardware.

2.2.7 Remote Debugging

Experience with earlier systems has convinced us of the advantages of including debugging support even in the lowest layers of an operating system. Unfortunately, many of the facilities needed to support a reasonable symbolic debugger (such as access to a file system) are not accessible to the low layers of an operating system, especially early in operating system development. We decided to investigate an

alternative approach for Swift: a *remote debugger*, in which most of the code and all of the intelligence of the debugger are moved off the machine under debug and onto a development machine. The development machine can provide all the desired supporting facilities, such as a file system, easy access to symbols and source code, logging facilities, and so forth. The machine under debug, on the other hand, contains a very small "stub" of code to carry out the debugging requests generated by the user on the development machine. The two machines are connected by a network of some description; in Swift this is the local-area network which forms the backbone of the entire distributed system. The remote debugger is known as RDB.

Several problems had to be solved in designing a remote debugging system. Although remote debugging protocols had been designed before, none was adequate for the job; so a new protocol had to be designed. In contrast to existing remote debugging protocols, RDB gives the user the capability to interrupt the execution of the program under debug at any time by sending an RDB request packet. This required tricky design in the remote debugger stub: the existing Swift network device driver had to be modified to watch for debugger packets and transfer control to the debugger at the appropriate time. The remote debugger stub then had to usurp control of the network device for the duration of the debugging session.

To date, the remote debugger has proved very useful in Swift debugging, particularly in finding problems related to synchronization and locking problems. As mentioned in the section on CLU, many of the typical problems arising in operating system implementations (such as dangling pointer problems) have been essentially eliminated by our choice of CLU as the systems programming language. Nonetheless, the remote debugger has proved worth the effort.

We envision a further use of RDB in performance measurement. In particular, we need to be able to gather statistical information on execution times and call frequencies, and then analyze this information. The analysis requires access to the symbol tables of the program being analyzed, and hence must be done on the development machine. We plan to use RDB for gathering the statistics and transporting the statistical information to the development machine for analysis.

2.3. Plans

We are still at a very early point in the development of Swift. At this point a preliminary implementation of the multi-tasking system and deadline scheduler is operational, along with a rudimentary memory management system, and the low-level support code required to run standalone on a VAX 11/750. Several device drivers are available, including a driver for the proNET token ring and an implementation of the DoD Internet protocol; the device drivers use the upcall model described above.

In the immediate future, plans call for a rewrite of much of the multi-tasking code, to reflect our improved understanding of the problem and increase performance. At the same time, we will begin using the Argus compiler being developed by the Computation Structures Group, which we expect will both improve performance and keep us on a closer track with the Argus implementation. Also in the near term, we will bring up a rudimentary, non-real-time garbage collector.

Longer-term work for the upcoming year will focus on the areas of: garbage collection; network support; file systems, including UNIX file system support and SWALLOW; and linking.

2.3.1 Garbage Collection

As explained above, an important goal for Swift is that it provide real-time response when needed, rendering unsuitable conventional garbage collection algorithms, which result in all computation halting while garbage collection is being performed. Several schemes have been proposed in the past for performing garbage collection in real-time; all, however, have been plagued by efficiency problems. Work is in progress on modified versions of Dijkstra's real-time garbage collection algorithm; we feel we have several promising approaches. We consider the design of a real-time garbage collector to be the most important task facing us in the next year.

2.3.2 Network Support

The level of network support currently provided by Swift is minimal, but the code already written is quite solid. Major tasks to be tackled in the next year are completing the implementation of the Internet protocol, including routing and error handling, and writing a version of the Transmission Control Protocol. The TCP will be a major test of the upcall model; its design will draw on previous TCP implementations done by our group for various machines.

An early goal is a version of the Remote Virtual Disk protocol designed by our group, which is presently providing remote disk access services for the Laboratory's VAXes. This is a necessary component of the file system projects described below. We also expect to soon be running an implementation of the BLINK protocol, providing remote access to bit-mapped displays and permitting work to begin on the Swift user interface.

Little effort has as yet gone into the design of the higher-level network services which will ultimately be needed. Such services as authentication and service finding will be supported in Swift through the network; much design work remains to be done in this area.

2.3.3 Filesystems and SWALLOW

Ultimately, we expect long-term data storage in Swift to be performed by the SWALLOW distributed data storage system. SWALLOW provides an object-oriented storage system, well suited to the object orientation of CLU; moreover, it solves the problems of concurrent access to shared long-term data and of recovery after crashes.

Work to date on SWALLOW implementation has been performed on UNIX and on the XEROX Alto's, as Swift is not yet suitable for supporting SWALLOW. We anticipate that it will take us some time to learn to use SWALLOW and to modify existing applications to take advantage of its features; until then, the ability to access files on a standard UNIX file system from Swift programs would be very valuable. Accordingly, we have begun an implementation of a UNIX file system for Swift. The implementation should support the basic file system operations of creating, opening, closing, renaming, and deleting files, and the basic file operations of reading and writing blocks of data. We hope that the implementation will be useful both for use with locally-attached disks and with remote virtual disks. It should provide us with the ability to manipulate and manage long-term data well before SWALLOW becomes operational, and thus should help support the development of other pieces of the system, such as the linker.

2.3.4 Linking

Ultimately, Swift is intended to be useful as a general-purpose computer system. As such, it must be possible to initiate new programs and to replace existing instantiations of routines with new or updated versions. In short, we need a linking facility which permits new programs to be brought into the system, and which permits unused programs to be garbage collected. Such a linker must be able to resolve references from the newly-instantiated program to already-instantiated modules in the system; it must also provide facilities for resolving references to other not-yet-linked modules and arrange for those modules to be loaded.

Linkers may be characterized in terms of how early or late they perform the binding between symbols and addresses. There are three general categories:

1) Static linkers. The key characteristic of a static linker is that the operation of resolving free references is separated from the operation of initiating a program; at the time a program is initiated into the system, it must not have any free references. The free references are bound by an explicit linking operation, producing an executable image in which all references are bound. Replacing a module in a program requires relinking the entire program.

2) Incremental linkers. In an incremental linker, free references are

resolved at the time the program is initiated. All free references must be resolved at program initiation time (although some references may be bound to "stub routines" which simply raise an error condition if they are ever called). To replace a module in a program, any current instantiations of the program are simply terminated and the program is initiated again. Note that this implies that at program initiation time, a "context" must be supplied to guide the resolution of references.

3) Dynamic linkers. A dynamic linker resolves each free reference only when the reference is actually used. When a program attempts to follow a free reference, a "dynamic linking fault" occurs, and the reference is resolved (with respect to some linking context, which must be avilable at run time). Replacing a module in a dynamic linking system is very similar to module replacement with an incremental linker.

A dynamic linking system is the most flexible and probably the most desirable; however, it generally requires special hardware support to run efficiently. An incremental linking system is almost as flexible and can run much more efficiently. We will attempt to implement an incremental linker for Swift.

There are a number of issues which have to be resolved before implementation of the linker can begin, including:

- The representation of programs and their static variables in memory.
- Interactions between the linker and garbage collector.
- The representation and usage of the linking contexts, which guide the linker in performing the symbol resolution.
- Details of the module replacement process, including the issues of redefining abstract data types.

3. SWALLOW DISTRIBUTED DATA STORAGE SYSTEM

During the past year, the most significant progress on SWALLOW has been the completion of a prototype broker by Craig Zarmer.

The SWALLOW broker is the software on each node of the distributed system that manages the store that belongs to that node. Such storage may be on a local disk, or remotely stored on a shared SWALLOW repository (data storage server).

Zarmer designed and built a prototype broker, running in CLU on top of the UNIX

operating system. The most interesting aspect of his work was the development of algorithms for extracting data from the repository in local primary memory. The cache management algorithms must be carefully designed so that if the node crashes, with loss of the data in the cache, the system properly recovers. Thus the cache manager takes into account the concurrency control and failure recovery algorithms of the SWALLOW system.

In addition to the broker implementation, Zarmer analyzed the performance improvements due to the cache. For many applications, Zarmer's cache will significantly improve performance, as compared with a cacheless broker.

4. NAMING FACILITIES FOR FEDERATED SYSTEMS

Karen Sollins has been working on questions of how people and computers use names and how computer naming can be brought closer to human naming.

Names form the basis of communication both among humans and between humans and computers. In order to communicate with another human, the human must be able to name objects and actions in such a way that both humans understand the names. Analogously, in order to communicate with a computer, the human must be able to name operations and objects in a way meaningful to both the human and the computer. Therefore, what can be named and how is a central issue in designing a computer system useful to humans.

Sollins' work is an investigation of a naming framework for a distributed computer system, using human communication patterns to provide a set of goals for the framework. The system model is one of a *federation* of loosely coupled computers connected by a communications network. The goals for the framework based on human communication, plus the constraints presented by the federated system model, will provide the basis for the technical problems to be addressed in her thesis. In addition, since the functions provided by this naming facility will be different from those functions provided in past naming facilities, the thesis must address how those additional functions will be provided for the users of such a computer system.

In the past, naming facilities in computer systems have been restrictive. The space of file names was likely to be hierarchical and the name on each branch of the hierarchy might be limited in length. The space of names identifying users might be flat or hierarchical and might be limited to a small number of characters. Processes, even subprocesses, often were only namable very awkwardly (perhaps by a number) if at all, even by a subprocess's parent. None of these has much in common with the way people name things, particularly when communicating with other people.

There are two reasons for naming entities, both having to do with communication. First, names may be used by an individual to organize and remember named entities; names provide a taxonomy. This sort of name is used by an individual or group to organize information. Second, names may be used among a group of people as the basis of communication. In order to communicate, the group must agree on the meaning of the names used. Over time, they may expand the set of names on which they agree. They will use certain protocols both to reach such an initial agreement and to expand further their basis of agreement.

The human clients of a computer system have been trained since early childhood in using a naming framework for communicating with other humans. A move toward imitation of the mechanisms used among humans would improve usability in the naming facilities provided by computer systems. The following seven observations about human use of names provide a basis for an improved computer naming facility.

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 contexts.*

4) **Flexibility of usage of names:** *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 in order to narrow the set of objects that are named.*

5) **Manifest meaning of names:** *The words used by humans for names have meanings constrained by human languages. These meanings are understood by other humans as well.*

6) **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.*

7) **Unification:** *Humans use only one naming system for all kinds of things.*

The direction in which computer systems have been moving has been toward a multiplicity of machines interconnected by networks providing a communication medium. The concerns of privacy and independence from other users have always been issues among computer administrators and users, but the nature of those concerns have changed somewhat as smaller, cheaper computers have become available. In many cases, administrators purchase such computers and put them into service in isolation. At some later time, the administrators decide to connect the computers under their management. From here, the collection may continue to grow with little control or consensus among the participants in such a "system." An *autonomous* computer is one for which all decisions are made independently of the decisions made for any other; all the activities on one computer are isolated from the activities of any other. Many administrators have pursued this option in order to escape large time-sharing systems. A *federation* is a loose coupling of computers to allow some degree of cooperation, while at the same time preserving a degree of autonomy. In a federation, there is some agreement on behavior and protocols to be utilized, but the barriers apparent in the isolated machine are still available to anyone who wants to enforce them. If the administrator or user wants to disconnect the computer from the network by simply not accepting messages, that is possible. If that computer provides a service to the participants in the network, they must understand that such a service will not always be available. On the other hand, federation provides the common ground for communication (such as agreement about protocols and services to be available) should it be desired. The loose coupling labelled federation is underlying system model of this research project.

Autonomy in the federated system limits the set of organizing structures it is possible to build. For example, sharing of information, such as collections of names, across node boundaries is restricted by the fact that the only means of communicating across node boundaries is by passing messages. The thesis will explore both the constraints from above (the clients) and the limitations from below (the federation of nodes), and will provide a naming facility conforming to those restrictions.

Briefly, the mechanisms proposed are based on two new types of objects, the *context* and the *aggregate*. A context translates names into entities. It can be given pairs of names and entities to remember and translate on demand. An aggregate is

a structured set of contexts. Each aggregate has a *current context* reflecting that part of the aggregate that is being actively used by all the participants in the communication and an *environment* reflecting the private information that a participant carries to the aggregate. The current context is a single context. The environment is a collection of contexts, possibly ordered. An aggregate is an individual's view of the name resolution facility available while communicating with others.

Further work will include implementing contexts and aggregates in order to further investigate their feasibility and utility in supporting human-computer naming requirements. On the other hand, an implementation that is not a complete user environment cannot investigate fully all the issues discussed above. The thesis will consider those issues in more depth than the implementation will allow. In addition, traditionally, some naming mechanisms have provided functions that are not provided by the mechanisms of contexts and aggregates such as authentication, protection, management of other information such as time of creation or last use, and many more. The thesis will also address the problem of supporting those functions that users expect from their naming facilities.

5. INTER-ORGANIZATION NETWORKING

During the past year we continued our efforts to understand the issues raised by network interconnection across administrative boundaries; Deborah Estrin has chosen this as the subject of her doctoral research under the supervision of Jerome Saltzer. We are pursuing three related lines of investigation, each of which has both technical and non-technical components:

- What are the policy requirements for network-interconnection technology when the interconnections span administrative boundaries? How must the technology developed for *intra*-organization use be modified to satisfy these requirements, e.g., network access controls, policy filters, authentication mechanisms? How do these requirements vary as a function of the application supported over the connection, e.g., electronic mail vs. remote login.

- What are the organization implications for external interconnection? How must the connecting organizations modify existing internal policies, procedures, and configurations, all of which were established under the assumption that internal resources and facilities would be accessible to internal users only? How do these organization implications vary as a function of the technical characteristics of the connection, in particular the degree of integration with internal facilities?

- What are the public policy implications of inter-corporate networking? What is motivating such interconnection, what industry sectors are involved, and how will this new form of inter-corporate relations affect

market dynamics, e.g., solidification of relationships between buyers and sellers, for example? What will be the role of public telecommunication services vs. private networks in providing the infrastructure for such interconnections?

Following an informal survey of the inter-organization networking activities that are currently underway (for example, transportation, grocery, insurance, airline, bank, pharmaceutical, university), we found that the fundamental difference between computer-communication networks that operate across administrative boundaries and more traditional inter-organization communication modes is that a user in one organization can cause some event to occur automatically within the domain of another organization, without any human intervention or auditing. Given this observation, it is useful to analyze inter-organization networks in terms of the application that is supported across the connection since the application determines the nature of event that a user in one organization can evoke in a second organization. Interconnection arrangements can be grouped into four categories of application -- electronic mail, database transaction, file transfer, and remote login. These categories differ from one another in the range of capabilities made available to external users and the degree of control over external usage available to each organization. The potential organization policy concerns intensify as the number of internal resources that the external user is given access to increases:

- Electronic mail is the most restrictive. It allows users to send and receive messages but not to extract any information from the remote system. Therefore, security concerns for the most part are limited to authentication of message originators and recipients to one another and to restricting overly burdensome volumes of undesired mail.

- Database transaction systems do allow extraction of information via querying, although typically the extent of interaction is highly restricted. Nevertheless, due to the *active* nature of such connections, security concerns include not only authentication of the remote user, but the checking of access rights as well.

- File transfer allows a remote user to extract or insert any file such as a program, data, or document. Therefore, security concerns extend to controlling access to all stored information.

- Remote login permits access to all system resources. Therefore, security concerns extend to controlling access to all system resources.

For the most part, these security concerns are the responsibility of the internal systems' security mechanisms and not of the communication facility. But, the presence of the more diverse, external community strains what were previously adequate internal security mechanisms and policies.

One class of policy enforcement mechanisms which might be applied to insulate

interconnected networks, and therefore organizations, from one another is policy filters in gateways. This is for the most part a technical fix but does require that the organizations explicitly define what their policy requirements are. The current technological basis for providing policy control between networks is almost completely non-existent. Today, whenever a packet of data arrives at the boundary between organizations it is difficult for any person or program to discover its purpose, since that purpose is buried in layers of protocols, and this packet may be only one of many that are part of a single activity (e.g., a file transfer or host-to-terminal communications stream). Present approaches fall into one of three categories, none of which provides both satisfactory function and satisfactory control:

1) Allow the packet to cross, and depend on the end points (i.e., host systems or users) to initiate only communications that meet policy constraints. This technique fails, for example, if network B finds that it can be used as a transit network between stations on network A and stations on network C. In such a case, network B gets no chance to exert any policy control.

2) Require that all protocols terminate at each gateway between networks. For every application, place a program at the gateway to act as a monitor and relay. Since the protocol is terminated, the underlying purpose of the connection is visible to the monitor, which can more easily enforce policy constraints. This approach is analogous to making a telephone call in which each party can talk only to an intermediate operator, who relays the conversation. While acceptable for some applications, delay and loss of special features cripple other applications.

3) Do not permit the connection in the first place. This approach provides conservative control, but is rather devastating from an application point of view. Given the fear of the alternatives it is probably the most widespread technique used today.

We encountered inter-organization networking activity in three arenas: industry-wide peer networks, customer-supplier arrangements, and university/research center networks. Of these three, the university/research center arena employs the most sophisticated technology and applications. We attribute the relative intensity of organization policy problems encountered in this arena to the degree of integration of each participating organization's internal facilities with its external-communication facilities. Similarly, we speculate that the absence of such integration in existing industry-wide and supplier-customer communication arenas partially accounts for the rarity with which organization policy problems have been encountered to date.[CSS Publication 4]

5.1. IBM Interconnection Project

Recently we embarked on an experimental project with IBM to study the policy requirements of interconnected organizations and to implement examples of such links. The proposed undertaking consists of two parts: implementation of a link between a M.I.T. and an IBM local network, and investigation and study of the policy requirements that arise as a result of this interconnection.

The initial testbed for policy research will be a link between gateways attached to the M.I.T. local area networks (largely DARPA-provided and connected to the ARPANET) and the IBM Corporate Job Network. This link will provide us with first-hand experience experimenting with policy control mechanisms that are acceptable to the interconnected parties, but minimize interference with the function and performance of the underlying data communication systems. The initial milestone of this project will be the following: electronic mail between authorized parties can be originated either within the ARPANET or the IBM network and terminate at the other network, with satisfaction as to policy control expressed by the M.I.T. network and ARPANET operators (i.e., Defense Communications Association) and by persons responsible for asset control within IBM. Subsequent activities will include experimentation with remote login and file transfer capabilities as well as the use of information services.

The initial design of the connection is as follows: Messages destined for IBM will travel from authorized M.I.T. users via the ARPANET and local networks to a VAX 11/750 that operates as the site of policy screening on the M.I.T. side of the connection (Don Gillies, a U.R.O.P. student, is implementing the policy-filter and mail-forwarding mechanisms for the M.I.T. half-gateway.) After authorization, messages will be encrypted and forwarded from the *policy-VAX*, to the so-called PC-gateway, and over dial-up telephone lines to the IBM half of the gateway. The PC-gateway is an LSI-11 which forms the interface between an M.I.T. local networks and 8 dial-up ports. The IBM half of the gateway will decrypt the message files, perform any policy filtering deemed necessary, and convert the message format into one suitable for distribution over their internal network. Mail transfer from IBM to M.I.T. will operate in a similar manner.

The encryption of messages serves to authenticate to IBM that the messages were processed by the M.I.T. *policy-VAX*, and vice versa; in addition, encryption provides some protection from message interception. We also require a packet-level mechanism to insure that all packets arriving from IBM are forwarded to the *policy*-VAX before traveling elsewhere on the M.I.T. networks. Jerome Saltzer, in conjunction with David Reed, Deborah Estrin, and David Clark, has specified a protocol whereby the IBM half-gateway will initiate a connection to an authentication server via the PC-gateway, before being permitted to forward packets onto the M.I.T. local network. Once the connection has been authorized, the PC-gateway must be

able to certify that subsequent packets are in fact originated by the entity that was initially authenticated. We will use a link-level protocol developed by David Reed to provide the necessary link-level authentication. This protocol encodes a *ticket* in the header of each packet (the ticket is agreed upon when the connection is first authenticated) to certify that the packet was originated by the entity that established the authenticated connection.

5.2. Network Access Control

Network Access Control is an important component of a solution to the problems of inter-enterprise communications. Network access control is our term for methods of limiting and accounting for traffic that enters a network to manage the network communication resource. That is, network access control is a way of controlling who can use a particular network, and, to some extent, controlling allocation of the network resources.

We assume that networks are interconnected by inter-enterprise gateways. Such gateways' primary job is to forward packets from one network to another. Our approach to network access control is to provide gateways with enough information to decide for each packet whether or not to forward.

An analogy is the international system of passports and visas. In this system, a person may cross a national boundary if he/she is in possession of the appropriate passports and visas. The border-crossing criterion is simple and fast to apply. Border-crossing policies, on the other hand, are implemented by individual countries through such agencies as consuls or embassies, which make a policy decision before supplying appropriate visas.

We have designed a system for network access control that resembles the visa system. [CSS Thesis 4]. Each gateway between enterprises logically combines two agencies, one for each network. When a packet arrives at a gateway, the agency for the source network will require an appropriate exit "visa" before forwarding. The agency for the destination network similarily requires an appropriate entry "visa."

Entry and exit "visas" must be difficult to forge. Our method uses a characteristic fraction computed using a reasonably secure cryptosystem such as DES (a so-called cryptographic checksum). At any point in time, the gateway knows a set of keys for use in computing such characteristic functions. To contain damage (due to lost keys, stolen "visas" etc.), keys are changed frequently in the gateway.

Corresponding to the embassies, there are Network Access Control Servers (NACS) for each network. In order to set up communication through a network, the source of messages must negotiate with each NACS for the networks its traffic must

pass through. This is done dynamically. A packet entering/leaving a network with a "null visa" is forwarded to the NACS for that network by the gateway. The NACS authenticates the source of the packet using some encryption-based authentication system such as that proposed by Needham and Schroeder.[1] If its use of the network is proper, then a key to generate "visas" is sent to the source of the packet, and the packet is forwarded on.

In order that the source can properly control the path of its packets, we strongly recommend source routing [2] at the inter-enterprise connection level.

Uses for this "digital visa" scheme include control of an enterprise's information assets (asset protection) by the use of exit visas, bill source for transit network usage using entry visas, managing audit trails at the NACS, etc.

6. COMMUNITY INFORMATION SYSTEM PROJECT

Since the Community Information System project started nine months ago we have constructed software that maintains an inverted full-text data base of articles from *The New York Times* and an electronic clipping service that performs selective dissemination of information. The performance and reliability of the system has now reached a point where members of our staff use the system in place of a morning paper.

The goal of the Community Information System Project is to investigate ways of using computers to help people communicate more effectively. Over the past year our emphasis has been placed on building a data base of interesting information to support our on-line browsing and clipping services, keeping in mind the data base requirements necessary for our extension of the service into laboratory members' homes next year.

The Community Information System consists of many subsystems which communicate over a wide variety of communication channels. Tracing the flow of a sample piece of information should help clarify the function of the system.

Our primary information source is currently *The New York Times* news service. The news service arrives from New York at Technology Square on a standard telephone circuit. At Technology Square the signal is de-multiplexed, converted to a standard EIA signal, and made available to software on a VAX/750 UNIX system (MIT-CLS) by special-purpose hardware.

On the UNIX system a dedicated process continuously listens to the news wire and accumulates articles. The process accumulates 6-level Baudot characters from the serial port dedicated to the news line, converts the Baudot characters into ASCII,

watches for article boundaries, and stores each article in a separate file in the UNIX file system.

The appearance of a news article triggers several events. First, a program called the parser converts the text into our standard information item format. Information such as the author, title, priority, and subject of the article is extracted and stored in standard headers. If the article was split into several pieces for transmission the parser recombines it. Once the article is in standard format, the parser moves it to an output directory for the next step of processing. The parser also uses the information it extracts from articles to maintain a synopsis data base which is used by the on-line browsing program.

Once the article has been converted to a standard format, it is included in a full-text inverted data base of on-line information items. An online tool called "browse" allows users to access articles by specifying a desired article's category, type, or by specifying free text keywords that appear in the article.

In addition to maintaining an on-line data base, users can send mail to "Clipping@MIT-CLS" to specify a standing query or "filter." After an article is processed by the parser it is examined by the clipping service. The article is mailed to any user who has submitted a filter that matches the article. Filters are boolean combinations of words and phrases that can be applied to the priority, author, and text fields of an article.

The data base system and clipping service we have built are general purpose, and are not limited to processing information from *The New York Times*. The design of the system is intended to make the addition of new sources of information as easy as possible. For example, we are currently finishing software that will allow users to make data base submissions via ARPANET mail. Once this software is complete, ARPANET bulletin boards will be included in our data base.

In addition to central site services, we are making our data base available to geographically dispersed computers. This is accomplished by broadcasting information on a low-cost digital packet radio system. Remote computers use the broadcast information to update their local data bases according to the interests of their owners. The scheduler for the broadcast channel is complete, as is the engineering work to start digital broadcasting this summer. By next year the software for the remote computers will be complete. The software will keep remote data bases up to date and display information according to priorities set by its operator.

An important lesson that we have learned this year is that text need not be indexed by hand to be useful. Full-text indexing of articles proved to be very effective in allowing users to select relevant sets of articles from the data base. Part of our

continuing interest lies in human engineering the system's user interface to permit non-professional users to use it as an everyday tool.

7. MAKING THE VAX LOOK LIKE A PERSONAL COMPUTER

We have purchased a number of VAX 11/750s from DEC. These are meant to be used for research in single user machines.

We firmly believe that personal computers as powerful and as large as the VAX will be available to the consumer within five years. These computers will fit on a desk-top and will be reasonably priced. We want to determine interesting ways to use these computers *now*, so that when the technology arrives, we will be prepared.

Simply using the VAX as a single-person computer does not make it a personal computer. Personal computers possess certain characteristics. A personal computer is always accessible, operates for you continously, and is configured to your personal taste. It does not have to be portable, but it should not be difficult to move it. The VAXes do not have these features.

The VAX is large, hot, and noisy, and is therefore not suitable for your office or your home. It is not accessible in the same way that *personal* computers are. It is possible to attach a line from your VAX to your office and maintain the illusion that the VAX is in your office, but it is not terribly easy to move the VAX to another office. Several drawbacks come about because there are fewer VAXes available than people who want to use them as personal computers. This means we must take turns using a VAX as our personal machine. Sharing personal computers presents several problems. Sharing prohibits continous operation on your behalf. It is not polite to simulate a circuit for several days if people are waiting to use the machine. If more than one person uses your "personal" computer, whose taste is it tailored to? If we are able to configure a computer according to people's tastes, does this then mean that they can only use the single computer that is theirs, even if several other VAXes are sitting idle?

We have performed research aimed at making the VAXes act like personal computers. This entailed maintaining the illusion that each person had a VAX in his or her office that belonged to them *personally*. We accomplished this by attaching the VAXes to a local area network[3]. Two research projects dealt with making the VAXes your "personal" computer. BLINK allowed you to attach a bit-mapped display with an input device to the VAX. This display sat in your office, and was connected to some network. RVD allowed you to attach your "virtual disk drive" to

---

[3]The Proteon proNET (also known as the version 2 ring). The proNET is a 10 Mbit/sec ring network.

your computer. The virtual disk drive is really only a segment of a large disk that everyone shares.

7.1. RVD

Our ideas about personal computers were influenced by our use of XEROX Alto's. Each Alto was identical. Each user had their own disk(s) that they inserted into the Alto's disk drive when they booted the machine. We tried to do something similar with the VAXes. The result was RVD - the Remote Virtual Disk protocol.

The goals of the RVD project were to find a way to allow you to approach any available VAX, "spin up" your disk, and have a personalized environment. Since the VAXes were located far from our offices, and RK07 disk packs were expensive, allocating an RK07 pack to everyone was unacceptable. RVD provides each machine with many "virtual" drives in which a user can "spin up" any of his disks.

Other advantages of remote disks are the ability to use machines that have no disk drives, the ability to obtain economy of scale by purchasing secondary storage in large chunks rather than an RK07 at a time, and to share common code, rather than duplicate it on everyone's disk[4].

Why did we implement RVD as opposed to a file server? RVD provides more flexibility. A file server imposes the clumsy model of files on its clients. Some of our systems have no notion of files. In general the concept of adding on some number of disk drives seemed much cleaner and natural than the complicated idea of sharing files, and retrieving them from some common source. Adding a drive to page off is understandable in a virtual disk system. Mounting a filesystem on top of a disk drive seems very straightforward. This allows you to slip in a shared file system underneath the operating system transparently. Once you have spun up a disk, it is attached to your computer. You do not have to negotiate with the remote server for each file access.

RVD consists of two halves -- the RVD server and the RVD client. The RVD server manages the disks, and responds to the clients requests over the net. The current server implementation is running as a user program on a VAX 11/750 running UNIX. It has 3 RA81's attached to it. The protocol was designed to minimize the computational overhead at the server so that it should not be a bottleneck. The RVD client is inside the UNIX kernel. We wrote a device driver for virtual disks, and it fits neatly under the UNIX operating system.

---

[4]The savings can be enormous. At M.I.T., a complete UNIX (man, sources, lisp, pascal compiler, CLU system, and so on) is larger than an RK07. A scaled down version still takes a substantial portion of the disk. Storing most of UNIX on RVD disks allows the full use of UNIX, and allows each of the 22 disks to be used for useful storage.

RVD has been in service for about 8 months. Until this month the server was running as a user program on a time-sharing system with a single RM80 used for RVD. From the client machine reads are comparable to an RK07, while writes take about twice as long. The current version of the server was a quick and dirty hack, written originally to run on a PDP 11/45. It was meant to test the client code. When the server is rewritten we expect to see write times comparable to read times.

Because of the limited disk space and the slow writes, we have limited RVD to be used for shared read-only disks, and for file system backup. A full UNIX can take as much as 75% of the disk space on an RK07. Most of UNIX is now run off of virtual disks. RVD has also been very useful for tape backup. Without RVD we would be required to shut our VAX off, and physically carry our RK pack to a machine that has both a tape drive and an RK07 drive. With RVD we just spin up a backup disk on both machines, and copy the appropriate dump to tape, while our system is running.

We have just brought up our RA81 drives. This has added more than a Gigabyte of storage to RVD. The writeable disks have just been allocated, so we should probably see an increase in usage of RVD. Currently, the RVD server receives about 500K packets every 2 days, and sends about 1M packets in the same interval.

8. AN ARGUMENT FOR SOFT LAYERING OF PROTOCOLS

During the year Geoffrey Cooper completed a Master's thesis concerning protocol layering. There are two ways of looking at what the thesis accomplishes. From one point of view, the thesis begins with the fact of layered protocols, analyses them, finds them lacking, and suggests a "better way" to write protocols. From a different point of view, the thesis develops the need for layered protocols, examines their advantages, and suggests how layering may be maintained in a protocol design without undue cost to the efficiency of the protocol implementation.

The thesis concentrates on one particular situation, that of a layered protocol architecture which implements reliable communications between cooperating application-level entities. In this context one sees that the maintenance of a layered structure in the protocol implementation could cause it to be so inefficient as to be unusable. After a good deal of analysis, an extension is introduced to protocol layering which provided a mechanism whereby the shortcomings of the layered structure can be fixed.

The thesis begins with a development of the concept of protocol layering, and an outline of the advantages of the scheme. This discussion notes that because there are typically many different network entities inside of a computer system, but only one (or perhaps two) hardware interfaces to the network, it is a requirement of the network software in the system that it allow all of these entities to share the network

hardware. Further, because the task of implementing all the network applications in a host is a major one, there is a strong desire to modularize the structure of the network software in a computer system in such a way as to make it possible for the different network applications to share, at least, part of the code that implements them.

Protocol layering provides an elegant means of satisfying these two criteria in a single mechanism. In a layered protocol architecture, each layer of the architecture provides to the layers above it a more sophisticated set of services than is provided by the network hardware. The nature of the "refined" service that is provided by each layer is such that the service fits into the requirements of many of the network applications. Some applications will wish to make use of this refined service directly, while others will make use of it indirectly through the device of added layers of protocol (each of which refines the service further). Protocol multiplexing can also be provided in each layer of the protocol architecture, to allow applications to use the layer's services directly without letting them interfere with transport layers which wish to further refine the layer's services. As a side effect of the introduction of protocol multiplexing, the requirement of being able to share the network software among many network entities is also met.

Protocol layering is, then, a powerful technique for achieving modularity of the design and implementation of network software. It has been central to the discussion of the preceding chapters that the advantages presented by protocol layering are sufficient that it would not serve to abandon a layered structure entirely.

Still, protocol layering is not without severe shortcomings. Because of the uncertainty which is inherent in all network communications, a network entity must always maintain a *death timer*, which provides it with a mechanism that it can use to avoid waiting forever for a cooperating remote entity which has failed. In a layered protocol implementation, the same problem occurs at every level of protocol, so that it is necessary for the implementation of each layered entity to provide its own death timer. Since, at a given time, the cooperation of all the layers in use is required for any useful communications to take place, it clear that all but the shortest of these death timers are really unnecessary. All but one of the death timers in a layered structure is thus a parasitic side effect of the introduction of protocol layering.

It is also common practise to set shorter *optimization timers* for the purpose of providing different transmission characteristics than are available in the lower layers of protocol. Since lower layers may piggyback messages with those of higher layers, an optimization timer at a lower layer which is longer than one at a higher layer is redundant. Thus, protocol layering encourages redundancy of optimization timers as well as death timers. The characteristic of layered protocols that caused their implementations to set many timers, most of which are useless at any given instant, was entitled the "timer problem."

A more severe problem than the timer problem is also investigated in the thesis. Entitled the *asynchrony problem*, it stems from the need for network entities to reliably coordinate state information with their cooperating remote entities. In a layered context, this coordination of state occurs independently at each layer of protocol, because of the perceived inability of layered protocols to coordinate this activity without violating their modularity. The asynchrony problem results in an increase in the number of packets sent over the network to perform a given function at the highest level of protocol. This increase is (in the worst case) exponential in the number of layers in the protocol architecture. Since many of the costs associated with sending and receiving packets are independent of their size, an exponential increase in the number of packets sent and received can be expected to result in a roughly exponential *decrease* in the relative throughput as seen at the application level protocol. It is thus a requirement of any protocol implementation that was to provide a useful service that it avoid the asynchrony problem.

There exist protocol implementations that *do* provide a useful service. The thesis examines some of the techniques that are used by these protocol implementations to avoid the asynchrony problem. These range from the total abandonment of a layered structure to a series of predictive "tricks" which work well for some higher level protocols, some of the time. The inherent harm in these techniques is that each is entirely independent of the protocol specification. Thus, to implement a usable version of a protocol, it ceases to be sufficient to simply implement its specification as written. If a protocol specification does not say how to implement the protocol, then its value is considerably diminished. Furthermore, the "tricks" needed to implement a protocol efficiently are generally not codified, and are often specific to particular protocols or operating systems.

The existing solutions to the asynchrony problem make clear the attractiveness of any solution to it that works *within* the context of a protocol layering, and is integrated into the protocol specification. The major effort of the thesis was to develop such a technique, which is called *soft layering*.

In a soft layered protocol, the protocol specification is augmented to include a "usage model" for the protocol: a model of the way in which the protocol expects higher level protocols to use it. Higher level protocols which conform to the usage model may expect to receive an efficient service from the protocol being specified. Other higher level protocols will still be able to make use of the service defined in the protocol specification, but the service provided to them will not be efficient. Soft layering provides a mechanism whereby the meaning of protocol efficiency -- which is *always* a part of the protocol implementation -- may be formalized in the protocol's specification. This ensures that all implementations of the protocol provide the same service from the point of view of both correctness and efficiency.

The analyses of protocols that were performed in the thesis led to another of its contributions. The thesis develops a remarkably succinct and useful terminology for analysing network entities: "happiness terminology." A network entity is said to be *happy* if it has received confirmation from its cooperating peer entity, indicating that every action requested of the peer has been completed (successfully or otherwise). It is *unhappy* if this is not the case: if there is some action which has been requested of the cooperating peer for which no confirmation has been received.

It is our belief that the concept of "happiness" is generally useful in the process of designing and implementing protocols, in a manner analogous to the way in which data abstraction is useful in the process of designing and implementing other kinds of software. For example, the question of how a protocol entity is made happy or unhappy is anlogous to the maintenance of a rep invariant in an abstract data type.

9. IBM PC NETWORK SOFTWARE PROGRESS

This project started one and one half years ago with the goal of making a personal computer act as a full-scale network host. The first step was to implement a file transfer program based on the Department of Defense InterNet family of protocols on an IBM Personal Computer. The second maior application was the remote login program, Telnet, based on TCP/IP, and on which much of the work of the past year has been done. Initially, the plan was to run the network protocols over a serial line to a gateway to the high speed networks. More recently, direct attachment to local area networks has been added.

9.1. Internet Implementation I

Last year's progress report described a very efficient file transfer package, TFTP, that achieved its effectiveness partly by being very non-modular. This year's goal was to insert modularity without losing effectiveness, so that a common internet layer could be used for both file transfer and remote login. The first step was to port the internet and UDP code from a UNIX implementation done by Larry Allen. We tried to preserve the software interface for the routines, but a major goal was also to prevent unnecessary copying of packet buffers between layers. Since the interface driver and the user program run in the same address space in our implementation on the PC, we were able to get away with copying data just twice: once into or out of the packet buffer for the user program, and once to or from the device.

The UDP- based name user code from the UNIX implementation was also ported with UDP. It resolves textual host names by polling known name servers over the network, and is integrated with all of the user packages.

We also needed a network interface driver. Anticipating a need for several different

network drivers, we modified the terminal emulator's serial line driver, gutting it and adding C code to deal with the link level protocol that we use over the serial line. We also required a serial line driver for the gateway that we used, so that packets generated by the PC could be forwarded to another network. We used Noel Chiappa's C-Gateway. David Bridgham wrote this driver, though its development was hindered by the rapidly changing hardware and software substrate (the C-Gateway was still under development at the time).

Louis Konopelski ported Larry Allen's TCP and Liza Martin's Telnet to the PC. This effort was simplified by the fact that our internet layer kept almost the same interface characteristics as the one written to work with TCP. The TCP uses a small non-preemptive multi-tasking package which allows it to be quite responsive to the asynchronous nature of the network. It also uses some of Dr. David Clark's ideas about upcalls.

We also decided that it would be good to have a TFTP which used the same internet library as the TCP, so we ported that TFTP from UNIX to the PC also, at negligible performance loss over the old one.

9.2. Internet Implementation II

We decided to modify the internet implementation to take advantage of the conditions on the PC under which it was running. On the UNIX system, it ran in a user process and communicated with the kernel via system calls. Here, with all the network code in a single address space and with our tasking system, we could do better than that, and have a consistent structure throughout the system based on tasks and upcalls.

The new implementation has a task associated with each network device. When a packet is received, the interrupt handling code enqueues the packet and wakes up the task. Later, when the task runs, it removes the received packet from the queue, does the processing of the packet that needs to be done at this level (protocol de-multiplexing) and calls internet with the packet if it is an internet packet. Then internet does its processing and calls TCP, or UDP, or ICMP, or GGP in turn with the packet.

When a layer wishes to send a packet, it fills in the parts of the packet that it wants to fill in and then calls the layer below it with the packet. Finally, the internet layer routes the packet, looks up the address of the routine to physically transmit the packet, and calls it.

This modification required an almost complete rewrite of the internet and UDP layers, as well as the introduction of ICMP and GGP code. (The only GGP function

supported is an Echo server). TCP was quickly modified to utilize the new structure, and it worked well.

January saw the release of a new network interface, the 3COM 10Mb Ethernet interface. We had to develop a driver for it for the PC, done by John Romkey, which slipped in modularly in place of the serial line driver, and also one for the C-Gateway, ported from BBN code by David Bridgham. In addition, we needed some way to translate from internet addresses (which are 32 bits long) to Ethernet addresses (which are 48 bits long). To do this, we chose the Internet standard Address Resolution Protocol, which also required implementation on the C-Gateway.

The new structure of the system proved itself the first time we tried an Ethernet Telnet. After resolving some differences in the implementation of the Address Resolution Protocol between the PC and the gateway (this was the first time they had ever spoken to one another), and one bug fix in the PC code, we had a working Ethernet Telnet.

At this point, further development was done on the Ethernet driver and much time was spent refining TCP and Telnet. We also wanted to bring TFTP up on the Ethernet, but the implementation which we were using would not port easily to the new internet, nor could it easily utilize a new driver. A new TFTP was then written from scratch. A TFTP to floppy disk typically has transfer rates around 15 Kbit/sec; TFTP's which discard the incoming file have run as high as 98 Kbit/sec. These TFTP's were with a PDP 11/45 running UNIX, and were done via the C-Gateway.

As we used the programs which we developed more and thought about the possibilities of having them run at other sites than M.I.T., we encountered several issues which caused us to build a customizer. The issues included

- having a single program run at different speeds on the serial line
- determining the PC's internet address on an Ethernet
- determining the addresses of name and time servers
- initial values in the internet to Ethernet address translation cache
- setting up personal attributes to Telnet (such as whether the back arrow key is delete or backspace)

There is a data structure in a well-known place in every program that contains initial values for these attributes and others such as what the program is, the version number of the program, and when it was last customized. Every program uses the same data structure, and the customizer simply allows editing of this structure

through a menu-oriented user interface, as well as duplication of the structure of another program. These changes previously required recompilation of the program.

A number of other programs were also developed. They include TCP *whois*, which queries a remote site for information about one of its users; *ping*, which sends out ICMP echo requests; *setclock*, which queries a set of time servers and sets the PC's clock based on the results; *hostname*, which resolves textual host names into numeric addresses and prints the addresses and the names of the servers which responded to the request; and *cookie*, a program which prints a "quote of the day" after having fetched the quote from a *cookie server*.

9.3. The Terminal Emulator

The terminal emulator was originally developed by David Bridgham to allow PC's to be used as terminals during program development. It emulated a DEC VT52 at the time. It was later upgraded to emulate a Heath H19, with the exception of ANSI mode and certain things such as keyboard locking which would be impractical on the PC.

During the development of Telnet, we realized that it would be much more useful to have the PC appear to be a smart terminal such as an H19, rather than as a dumb terminal which could do no cursor or screen manipulation. We decided to slip the same low level terminal emulator code in the standard I/O library terminal output code. This involved breaking the emulator up into two distinct parts, which handled the user interface, serial line, and actual emulation.

Experience with Telnet later showed that at times, Telnet could receive data for the screen faster than the emulator could handle it; the emulator became a bottleneck. This problem was rectified by having the hardware do the scrolling instead of the processor, as was the case before. With the improved emulator, Ethernet Telnet often performs better than a 9600 baud line wired directly to a machine.

We also found it necessary to add handling of some ANSI mode features since EMACS on one of the machines that many people around the lab use utilizes ANSI mode operations.

9.4. Remote Logging Protocol

To aid in debugging of a variety of programs, we implemented the remote logging protocol described by Dr. David Clark. Use of this service allows us to monitor machines that are in service, discover the reasons for failure of machines, and see when obscure conditions which should never occur do occur. A server for the logging protocol was done for VAX UNIX in CLU by Mark Rosenstein. User logging code was written for the C-Gateway, the IBM PC, the M.I.T. TFTP Dover Spooler, and

the Network Monitoring Station by David Bridgham, John Romkey, Geoffrey Cooper, and David Feldmeier.

10. INTERNET PROTOCOL WORK

10.1. Protocol Performance Improvements

Work continued this year on testing techniques for achieving higher performance from the Internet family of protocols on different types of computers and different operating systems.

10.2. UNIX Kernel Network Support

Many of the Internet protocols were implemented on our PDP 11/45 running version 6 UNIX. Since the kernel in this system has a small address space and a poor debugging environment, only the the most basic networking functions were included in it. In the kernel are modules to drive a proNET ring device, to transmit and receive Internet packets, to de-multiplex incoming packets, including UDP and TCP packets, to reassemble Internet fragments, to maintain a cache of Internet hosts and their best first hop gateways, and to route a packet to its appropriate first hop on the local net.

10.3. Application Processes

Outside the UNIX kernel are network application processes to handle remote login, file transfer, mail transfer, and network diagnostic, error and routing reports. The user and server Telnets deserve special mention since they run on TCP which is the most complex of the internet protocols.

10.3.1 Server TCP/Telnet

Server Telnet, the remote login protocol, runs in the same process with its TCP and IP layers; it supports one Telnet connection. Multiple server Telnet processes run simultaneously when several remote logins are being supported. Collapsing server Telnet into the same process with its TCP and IP has several performance benefits - such as eliminating interprocess communication and data copying, and decreasing the number of processes scheduled. It also allows TCP to query Telnet about any data it might want to send out with TCP connection maintenance information; this reduces the number of packets transmitted on a connection. The jobs of the various layers can be performed when most appropriate rather than when each protocol layer is "scheduled." This implementation of TCP supports most features of the protocol and includes code to prevent "silly window syndrome."

10.3.2 User TCP/Telnet

Network protocols are often specified with a large amount of internal asynchrony; this greatly complicates their implementation. This is particularly true in systems like UNIX, in which processes cannot share memory or communicate cheaply. The result is that protocol implementations often use special-purpose multiplexing to simulate asynchronous activities; this muddies the structure, making understanding and modifying the code difficult.

We explored a different approach with our implementation of user Telnet. We designed a small subsystem permitting multiple asynchronous activities ("tasks"), each with its own stack and machine state, to run within the context of a single UNIX process. Tasks can be scheduled in response to events external to the process, such as the arrival of a packet from the network, and by other tasks within the same process. A non-preemptive scheduling algorithm is employed to avoid coordination problems in accessing shared data.

The user Telnet includes a small implementation of the Transmission Control Protocol. This implementation was actually a translation of a TCP written in BCPL by David Clark for the XEROX Alto minicomputer. The TCP implementation uses three tasks, of which two are contained in the TCP module and one deals with timer management. One TCP task handles input packets; it is awakened by receipt of a signal from the network driver indicating that an input packet is available. The other TCP task handles packet transmission; it is awakened by the TCP receiver task (to send acknowledgments), by the user of the TCP (to send data), and by timer expiration (to perform retransmissions). The TCP tasks communicate with each other by sharing state variables, while communication with other protocol layers is by means of subroutine calls. A fourth task runs the actual Telnet implementation.

10.3.3 Trivial File Transfer

Due to the size and complexity of the full ARPANET File Transfer Protocol, we have not yet completed an FTP implementation; instead, we chose to implement the Trivial File Transfer Protocol (TFTP), a simple file transfer protocol built directly on datagrams rather than on a stream connection. This simple protocol has proven very durable and useful in the past; in addition to file transfer service, it is used for remote printer access, network bootloading, and has been used for mail transport.

It should be noted that, despite its relative simplicity, TFTP exhibits in microcosm most of the implementation difficulties found in network software in general. It is significant to note that the current implementation of TFTP on our UNIX achieves roughly three times the throughput of our previous UNIX implementation while occupying roughly half the space; this suggests the effects of the learning curve in network protocol implementations.

10.3.4 Simple Mail Transfer

The simple tasking TCP designed for user Telnet also forms the basis of the Simple Mail Tranport Protocol implementation. Some extensions were required to add "server" functionality, but the modifications were small.

10.4. Results

We have seen as many as eight active remote login sessions at once with reasonable response times. Some performance measurements have shown the following results. Round trip time for an Internet Control Message Protocol time stamp packet sent from the PDP 11/45 to itself took between 30 and 40 milliseconds; this required two packet transmissions and two receptions. The maximum TFTP transfer rate that has been observed was 133 Kbits/sec between our machine and a VAX on the same ring net. The TCP used in user Telnet has been observed to sink data in a memory-to-memory transfer from a VAX at 215 Kbits/sec. The server Telnet's TCP has been seen to send data in a memory-to-memory transfer to an Alto on an Ethernet at 300 Kbits/sec; the gateway between the proNET and the Ethernet was an LSI 11/03 running our C-Gateway code.

10.5. Gateway Implementation

10.5.1 Exterior Gateway Protocol

M.I.T. has been participating in the development and implementation of a new protocol to be used to pass routing information between systems of autonomous gateways. The protocol is called exterior gateway protocol (EGP); its purpose is to provide a more controlled method of passing routing information between gateways who may or may not trust each other.

A loose definition of autonomous gateways is that they belong to the same administrative organization, such as M.I.T., and are fairly homogeneous. The gateways within an autonomous system will use their own conventions for passing routing and up/down information among themselves, and will use EGP to pass routing information between themselves and the outside world.

10.5.2 Summary of C Gateway Progress

Copies of the C-Gateway were sent to Stanford and BRL. These installations now seem to be running quite reliably.

A lot of effort was put into making the C-Gateway more robust; about two dozen bugs were isolated and fixed, and logging to a VAX at startup time was added to monitor gateway crashes. It appears that the MIT-GW crashes about 5 times a day. Most of these crashes are soft crashes in that the machine just restarts without needing to reboot itself.

Better routing mechanisms were added or worked on such as a default gateway for packets destined to a net to which we have no route, ICMP redirects, and EGP.

So much work was done on the C-Gateway this year that the easiest way to describe it is with excerpts from monthly progress reports.

1) August and September 1982

> Much has happened on the C-Gateway over the last several months. It is now in full operational service in the main M.I.T. ARPANET gateway, and will shortly go into service at Stanford in the ARPANET gateway there and in several local network gateways at M.I.T.
>
> Substantial work has been done on the internal structure to speed it up even further, and a fast and simple general tasking mechanism with priorities has been implemented to allow finer control over internal work scheduling. Extensive analysis of operation in high-throughput applications internal to M.I.T. is proceeding; some preliminary results have already resulted in improvements. In one test at M.I.T. a while back, an LSI-11 was able to maintain a data rate of 1/2 Mbit/sec from a proNET to a 3MBit Ethernet; this is quite good considering the slowness of the LSI-11 and the fact that each packet must be byte-swapped before being sent out on the Ethernet. Indications are that a faster processor with two DMA interfaces could sustain data rates in the several megabit range. Some initial investigation of data flow and flow control inside the gateway has been done, and further modifications to allow more control in this area are planned.
>
> The code to handle PUP was written; it is now relatively complete, providing full simple gateway service, which is to say complete routing and ECHO but not name resolution or boot servers. The CHAOS protocol code was redone, and is now considered complete at the same level of simplicity (CORUT, STATUS and PULSAR). Work to expand the IP implementation to a full IP (including ICMP, GGP, class A/B/C support, etc.) has started; this was delayed until the previous two protocols were done to allow the experience gained there to be included. The ARPANET driver has been cleaned up, and plans to expand it to include per-link flow control are complete. Finally, an extensive audit of the code has been made to remove places where, in the initial rush to get the code up, bughalts were placed on unlikely code branches.

2) October 1982

As a result of analysis done last month, some major changes were made to the structure of the gateway software to allow better internal flow control as well as packet buffer reclamation. All packets are now kept on explicit queues instead of implicit ones (i.e., system message queues, etc.), and internal flow control code was installed. No code for external flow control (Source Quench ICMP packets) has been installed because the algorithms are still under consideration, but all the necessary hooks exist. Memory allocation was speeded up by keeping an internal list of free buffers instead of using the MOS memory allocation scheme. A smarter buffer allocation scheme (using loose pools and minimum reservation strategies) was installed.

The code has been deployed in additional sites at M.I.T., including the main gateway between the high-speed local networks at Tech Square, where it performs adequately. With the addition of a MOS device driver for the ACC ARPANET interface, obtained from SRI, a C-Gateway was configured for Stanford University to interface between the ARPANET and the collection of Ethernets there; this came up with almost no problems and packets were exchanged between a VAX on an Ethernet at Stanford and a VAX on a proNET at M.I.T.

3) December 1982

The C-Gateway code was packaged and shipped to the Army's Ballistic Research Lab in Maryland for installation in a gateway there. BRL will be writing additional device drivers and handlers as needed.

4) January 1983

Efforts are being made to improve the reliabilty of the C-Gateway and to collect more information about crashes. Software bughalts have been changed to save information about the crash and then to restart immediately. Also when the gateway starts up, it now sends a log packet to a log server on a VAX on the proNET ring; this log packet includes the message that was stored away by the last crash. Currently log packets are not always getting to the log server; this may be because when the gateway starts up, it reenters the proNET ring causing some perturbation in the ring.

Code to respond to ICMP pings was added to the gateway. Code to generate ICMP redirects when needed was also added. Due to fears about possible adverse effects on ARPANET hosts, the gateways presently send redirects only to hosts on the LCSnet.

5) February 1983

A driver for the Interlan 10Mb Ethernet interface was written and installed in the IBM PC gateway. It uses David Plummer's Address Resolution Protocol (RFC 826) to translate 32 bit internet addresses into 48 bit Ethernet addresses.

Slightly improved routing code was added to the gateways. Packets destined for nets to which a gateway has no route are now forwarded to a default gateway. This is a temporary solution to the Class B/C net problem. As a result, hosts on the LCSnet can access all Class B and C networks.

The version 1 ring network code and interfaces were removed from our gateways which means that this network has been decommissioned.

6) March 1983

The strategy for resynchronizing with the IMP when the IMP goes down has been changed to be similar to the strategy used by the BBN gateways. The old strategy was not robust in the cases where the gateway thought the IMP went down, but in fact the IMP was still up.

Ron Natelie at BRL-BMD is running an old version of the C-GW code that he has modified somewhat. He found a bug in the ACC driver that causes the gateway to think the IMP went down. His fix to this bug is included in the current ACC driver running on MIT-TGW.

Most of the code to implement EGP was written this month. As one would expect with an early implementation, the implementing process has turned up suggestions for some fairly minor modifications to the protocol.

7) April 1983

Several releases of the C-Gateway were sent to Stanford University this month. Some bugs were isolated and fixed; currently this gateway is providing a barely acceptable level of service. Debugging continues.

The BRL-Gateway version of the C-Gateway is running as a regular service now.

8) May 1983

Efforts on the part of Jeff Mogul and Bob Baldwin to debug the version of the C-GW running at Stanford turned up various packet length bounds problems. Jeff noticed that better packet length checks were needed in the Ethernet device driver and Ethernet network handler. When these checks were installed, the Stanford gateway became substantially more reliable. It now crashes about once every two days.

Earlier in the month Jeff and Bob figured out that a problem that looked like an IMP synchronization bug was in fact due to a feature of the ACC DMA interface to the IMP. The ACC device transfers two garbage bytes into memory at the end of the received packet. The extra bytes cause an input overrun when the gateway receives a maximum-sized packet. The device driver did not check for the overrun, so the garbage bytes would appear at the beginning of the next packet, causing it to be discarded due to bad ARPANET header format.

Our implementation of EGP was tested to itself over the M.I.T. test gateway's two interfaces (ARPANET and proNET) and to itself via an Echo server. It was also tested to DCN6 and DCN1. These tests have shaken out a few bugs; EGP seems to run quite robustly in the test gateway now.

M.I.T. took receipt of a Bridge Communications 68000 based computer this month. It will be used to develop experimental gateways.

11. NETWORK MONITORING STATION

In the field of local area networks, two types of networks dominate: an Ethernet type bus network and more recently the token ring network. Much of the current interest in rings seems to be due to the pending announcement by IBM of its token ring network. Although there is much debate over the relative merit of both types of networks there exists little information on the performance of rings. John F. Shoch and Jon A. Hupp of the XEROX Palo Alto Research Center produced a preliminary report on Ethernet performance containing such things as number of packet errors, performance under high load, stability and fairness. Unfortunately, no such comparable document exists for a token ring network. The purpose of the Network Monitoring Station is to collect statistics on the LCS proNET ring.

The Network Monitoring Station currently consists of a PDP 11/20 mini-computer with a network card for the proNET ring and some specialized hardware. The network cards for the proNET ring have two parts. The first is a Control Card which interfaces to the network on one side and has a standardize interface on the other.

The second board, the Host Specific board (HSB), has this same standardized interface on one side and a host specific interface on the other. The special hardware is placed between these two cards. The specialized hardware has three major components. The first is a HSB emulator that interfaces with the Control Card. The second is part is a Control Card emulator that interfaces to the Host Specific Board. The third part is a 32 bit, 40 microsecond resolution crystal clock.

The HSB emulator is the simplest. It always appears to be an HSB that is ready to accept a packet. No matter what the state of the true HSB, it will always receive a packet from the net. The Control Card itself is set in the "match all" mode which simply means that any packet that comes by will be received no matter what its address. The HSB emulator counts the number of incoming bytes and also keeps track of error signals received from the Control Card.

The Control Card emulator acts as a Control Card that only receives 17 byte packets (actually the first 17 bytes of an incoming message). By only working with the first 17 bytes of a packet, the Monitoring Station obtains the necessary information from the IP protocol and allows the HSB the maximum amount of time to reset for the next packet. The Control Card emulator keeps track of whether or not the HSB has received the incoming packet.

The clock is used for timing events on the ring and for maintaining the current time of day. Other things that are monitored include the times that the ring crashes and reinitializes, bad format packets, and hopefully soon, packets that are refused (not received by the addressee). The specialized board is memory mapped into the PDP 11/20 and also has an interrupt mechanism for fast retrieval of packet information.

The software running on the PDP 11/20 does some data compression, simple data accumulation and analysis. The only two items on the bus that can interrupt are the HSB and the specialized hardware card. These interrupts are fast (~20 microseconds) and simply place data from on board registers into a circular buffer. From here, the information in these buffers is analyzed when there is time. The software accumulates statistics such as number of packets and percentage of netload over the previous day, hour, minute and second. Also displayed is the time since last network crash (loss of token). Both network errors and monitoring station errors are accumulated.

Because the Monitoring Station has little storage or processing power, it would be desirable to get some information to a larger computer, perhaps with a tape drive for long term storage. Currently, the Monitoring Station compresses all of the IP headers down to protocol, ring destination, ring source, length of packet and time of reception at a factor of 64:1 and sends these packets over the proNET ring to a VAX. This VAX can do much more complex analysis than can the PDP 11/20. Although sending compressed information to another computer seems like a reasonable idea,

it might be better not to send packets over the network being monitored. An alternative would be to have the PDP 11/20 use either a serial line or a different network in order to transfer information. The best idea might be to have another mini-computer attached to the PDP 11/20. The PDP 11/20 would run the same software on a continuous basis. The other mini-computer could be used for real-time debugging and short-term network analysis running whatever software is useful at the time. At the same time, a tape drive would store all of the data generated by the PDP 11/20 on tape so that a complete set of records exist for later analysis. This later long-term analysis could be done on a VAX and since complete records exist, could be done any time any new information was required.

With the information collected, it is hoped that various parameters of ring operations and usage can be determined. Some of these include latency until transmission, number of defective packets, ring reliability, fairness and stability under high-load. Questions involving ring usage are protocols used, who is sending to whom, number of back-to-back packets, interpacket arrival time, distribution of packets size and distribution of usage throughout the day.

Currently, the Network Monitoring Station is running reliably but it still has some bugs in hardware and software that distort some of its statistics. Also a very elementary analysis program exists on the VAX for long term analysis. The proNET ring presently carries about 850 thousand packets and 140 million bytes on a busy day.

12. DIRSYS: AN ONLINE DIRECTORY ASSISTANCE SYSTEM

12.1. Overview

DIRSYS is an electronic telephone book. It was developed for users with widely varying computer skills. Therefore, the self-teaching aids for the novice were designed to not encumber the experienced user. It is based upon the familiar concepts of a paper phone book and a full-screen display editor such as EMACS. Entries from the directory database are displayed on the screen in a compact format, one line per entry, and DIRSYS indicates which is the current entry of interest by emphasizing the entry's line (capitalize all letters, filling in empty fields with periods, displaying in reverse video, etc.) The user may direct the system to emphasize another entry (i.e. move the system's pointer) by issuing commands, similar to EMACS' cursor motion commands, or by searching for a name. The search mechanism is incremental. That is, after each character typed by the user, DIRSYS updates its pointer and the terminal screen, if necessary, such that the pointer rests on the entry whose name string most closely matches what the user has typed so far. A help facility, in the form of a menu, is provided to guide the novice user and remind

the experienced user what commands are available. The help facility operates in the same manner as the incremental interface, except the search mechanism has been removed. The default screen allows approximately a full screen's worth of entries to be displayed, each entry occupying one line of the terminal screen. All information concerning a particular entry cannot be seen using this compact format. The user may request DIRSYS to display much fewer entries on the screen and show each entry in detail. A command is available to switch between these two display formats. All commands retain their semantics regardless of the display format.

12.2. Current State of the Project

A prototype has been implemented on a DECSYSTEM-20 and is being moved to a VAX 11/750. The interface and database structure are to be evaluated and modified based on the evaluations. A mechanism for updating the database is being implemented.

References

1. Needham, R. and Schroeder, N., "Using Encryption for Authentication In Large Networks of Computers," Communications ACM 21,12 (December 1978),993-999.

2. Saltzer, J., Reed, D., and Clark, D., "Source Routing for Campus-Wide Internet Transport," Local Networks for Computer Communications, West, A. and Janson, P., Editors, North-Holland Publishing Company, Amsterdam, 1980. 1-23.

COMPUTER SYSTEMS AND COMMUNICATION

Academic Staff

J.H. Saltzer, Group Leader
D.D. Clark
F.J. Corbato
D.K. Gifford
M.V. Wilkes

Research Staff

L.W. Allen
S.T. Berlin
J.N. Chiappa
M.B. Greenwald
E.A. Martin

Graduate Students

R.W. Baldwin
G.H. Cooper
D.L. Estrin
J. Frankel
K. Koile
C. Lamb
L. Zhang

Undergraduate Students

D.A. Bridgham
D.C. Feldmeier
J.K.T. Genka
D.W. Gillies
C. Hornig
F.S. Hsu
F. Huettig
R.W. Hyre
E.R. Juncosa
D.J. Karlson
L.J. Kaufman
F.H. Klein
L.J. Konopelski
B.C. Kuszmaul
J.R. Lekashman
A. Madhaven
R.D. Osgood
M.A. Pinone
C.S. Rittenberg
J.L. Romkey
A. Rosenstein
J.M. Roth
H.J. Shinsato
S.D. Trieu
C.M. Zeitz

Support Staff

S.C. Comfort
D.J. Fagin
N. Lyall
M.F. Webber

The work of the Computer Systems and Communications group and the Computer Systems Structure group this year was so closely related that a single report best describes it. The single report will be found as a separate section in this annual report.

References

Publications

1. Clark, D.D., "Internet Protocol Implementation Guidelines," Internet Protocol Implementation Guide, Network Information Center, SRI International, Menlo Park, CA, August, 1982. Comprised of:

   - Window and Acknowledgement Strategy on TCP (RFC-813)
   - Names, Addresses, Ports and Routes (RFC-814)
   - IP Datagram Reassembly Algorithms (RFC-815)
   - Fault Isolation and Recovery (RFC-816)
   - Modularity and Efficiency in Protocol Implementation (RFC-817)

2. Corbato, F.J., "Time Sharing," Encyclopedia of Computer Science, Anthony Ralston, Editor, Second Edition, van Nostrand Reinhold Co., New York, 1983.

3. Corbato, F.J., "An M.I.T. Campus Computer Network," Campus Computer Network Group Memorandum Number 1, July, 1982.

4. Estrin, D.L., "Inter-organizational Networking: Stringing wires across Administrative Boundaries," Eleventh Annual Telecommunications Policy Research Conference Proceedings, (V. Mosco, Editor), Ablex Publications, Norwood, N.J., 1984.

5. Saltzer, J.H., Pogran, K.T., and Clark, D.D., "Why A Ring?" Computer Networks 7, (July, 1983).

6. Saltzer, J.H., Reed, D.P., and Clark, D.D., "End-to-End Arguments in System Design," to be published in Transactions on Computer Systems.

7. Sirbu, M., Estrin, D.L., "Cable Television Networks as an Alternative to the Local Loop," Proc. IEEE International Conference on Communications, June, 1983.

Theses Completed

1. Cooper, G.H., "An Argument for Soft Layering of Protocols," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

2. Genka, J.K.T., "A Dial Up Packet Switcher for an Internet Gateway," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA. May, 1983.

3. Hornig, C., "A Second Generation Network Interface for Multics," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

4. Hsu, F.S., "Design of a Human Interface for an Online Directory Assistance System," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

5. Juncosa, E., "A Simple UNIX File System for the SWIFT Operating System," S.B. thesis, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

6. Klein, F.H., "Selective Dissemination Service for Users Within a Computer Net," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

7. Konopelski, L.J., "Implementing Internet Remote Login on a Personal Computer," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

8. Pinone, M.A., "A Selective Dissemination Service for Users Within a Computer Net," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

9. Rao, R.B., "The Design and Implementation of a Mail System for Interlisp-D," MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, August, 1982.

10. Rittenberg, C.S., "AutoMMS: A System for Automated DEC/MMS Description File Construction," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA. May, 1983.

11. Roth, J.M., "Data Capture: Forms That Use Constraints," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

12. Roush, P., "Computerized Scheduling of Intramural Sports," S.B. thesis, MIT, Department of Electical Engineering and Computer Science, Cambridge, MA, July, 1982.

13. Trieu, S.D., "A Transmit System, the Scheduler, for the Community Information System," S.B. thesis, MIT, Department off Electrical Engineering and Computer Science, Cambridge, MA, May, 1983.

Theses in Progress

1. Feldmeier, D.C., "Performance of the Version Two LCS Ringnet Local Area Network," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, January, 1984.

2. Koile, K., "The Design and Implementation of an Online Directory Assistance System," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, September, 1983.

3. Lamb, C.W., "A Screen Oriented Data Base Editor," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, August, 1983. (Also S.B. thesis).

4. Lekashman, J.R., "Performance Evaluation of a Packet Switching Internetwork Gateway," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, September, 1983.

5. Osgood, R.D., "Implementation of File Transfer Protocol on UNIX and IBM-PC," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected date of completion, September, 1983.

Talks

1. Clark, D.D., "Internetting Local Area Networks," Conference on Local Area Military Networks, Griffis Air Force Base, New York, September, 1982.

2. Clark, D.D., "Protocol Implementation and Design: Practical Considerations," SIGCOMM 83, University of Texas, Austin, Texas, March, 1983.

3. Saltzer, J.H., "Communications Requirements for Distributed Systems," Series of lectures, Nippon Electric Company, Tokyo, Japan, January, 1983.

Committees

1. Clark, D.D., MIT Network Working Group

2. Clark, D.D., DARPA/TCP Working Group (Chairman)

3. Chiappa, J.N., DARPA/TCP Working Group

4. Corbato, F.J., CS Net Policy Support Group

5. Corbato, F.J., Advisory Committee for Health Sciences Computing Facility, Harvard School of Public Health.

6. Corbato, F.J., National Research Council: NBS Panel for Scientific Computing

7. Corbato, F.J., National Science Foundation: Review Panel for CS Net

8. Martin, E.A., DARPA/TCP Working Group

9. Saltzer, J.H., DoD/DDRE Security Working Group Member

10. Saltzer, J.H., Chairman, 9th ACM Symposium on Operating Systems Principles

11. Saltzer, J.H., MIT Network Working Group

COMPUTER SYSTEMS STRUCTURES

Academic Staff

D.P. Reed, Group Leader

Research Staff

M. Greenwald

Graduate Students

W. Gramlich
K. Sollins
P. Ng
J. Stamos

Undergraduate Students

R. Allen
R. Harteneck
G. Hopkins
R. Kukura
J. Leschner
J. Mracek
S. Routhier
N. Shafer
E. Siegel
S. Subramanian
T. Tran
J. Woods
C. Zarmer

Support Staff

S. Comfort

Visitors

Ø. Hvinden

The work of the Computer Systems Structures group strongly overlapped with that of the Computer Systems and Communications group. Consequently, the work is reported in a joint section. See that section for details.

REFERENCES

Publications

1. Reed, D.P., "Implementing Atomic Actions on Decentralized Data," ACM Transactions on Computer Systems, Vol. I, No. 1 (February, 1983), pp. 3-23.

2. Ng, P. and D. Daniels, "Query Compilation in R*," IEEE Database Engineering, Vol. 5, No. 3 (September, 1982), pp. 15-18.

3. Ng, P., L. Haas, P. Selinger, E. Bertino, D. Daniels, B. Lindsay, G. Lohman, Y. Masunaga, C. Mohan, P. Wilms and R. Yost, "R*: A Research Project on Distributed Relational DBMS," IEEE Database Engineering, Vol. 5, No. 4 (December 1982), pp. 28-32.

4. Stamos, J.W., "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory," ACM Transactions on Computer Systems, conditionally accepted for publication, 1983.

Theses Completed

1. Kaufman, L., "Implementing a Distributed Debugging System", S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., June 1983.

2. Ketelboeter, V., "Forward Recovery in Distributed Systems", M.S. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., January 1983.

3. Margolin, B., "Extension of the Multics Library System", S.B., MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., expected date of completion: May 1983.

4. Mracek, J., "Controlling Network Usage by Encryption-Based Protocols", S.B. thesis, MIT, S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., June 1983.

5. Ostar, H., "An Automated Database Manual", S.B., MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., December 1982.

6. Routhier, S., "An Improved Authentication Server", S.B. thesis, MIT,

Department of Electrical Engineering and Computer Science, Cambridge, MA., June 1983.

7. Topolcic, C., "Ensuring the Satisfaction of Requests to Remote Servers in Distributed Computer Systems", M.S. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., January 1983.

8. Woods, J., "Integrating a Remote Bitmap Display in UNIX", S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., June 1983.

9. Zarmer, C., "Implementing a Swallow Broker", S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., June 1983.

Theses in Progress

1. Allen, R., "Validation of an Authentication Server Protocol", MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., expected date of completion: August 1983.

2. Harteneck, R., "A Drawing System in CLU", S.B., MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., expected date of completion: August 1983.

3. Gramlich, W., "Checkpoint Debugging", Ph.D., MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., expected date of completion: May 1984.

4. Ng, P., "Library Management in the Swift Distributed System", Ph.D., MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., expected date of completion: May 1985.

5. Shiroma, J., "Protocol in the Swift Operating System", S.B., MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., expected date of completion: August 1983.

6. Sollins, K., "Name Management in a Distributed System", Ph.D., MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., expected date of completion: May 1984.

7. Stamos, J., "Multi-Language Access to Persistent, External Data",

Ph.D., MIT, Department of Electrical Engineering and Computer Science, Cambridge, MA., expected date of completion: May 1985.

Talks

1. Reed, P., "Local Area Networks: A Research Perspective", Diebold Research Conference on Office Systems and Decision Support Systems, St. Paul, MN, July, 1982.

2. Reed, P., "The Swift Distributed System Testbed", MIT-IBM Mini Conference on Advanced Personal Computers, Lenox, MA, January, 1983.