SECURE RELIABLE PROCESSING SYSTEMS: FINAL REPORT

Gerald Popek

February 1984
Report No. CSD-840228

SECURE RELIABLE PROCESSING SYSTEMS

FINAL TECHNICAL REPORT

1 July 1977 - 31 March 1982

Gerald J. Popek
Principal Investigator
Computer Science Department
School of Engineering and Applied Science
University of California at Los Angeles
(213) 825-6507

This research was sponsored by the
Defense Advanced Research Projects Agency

ARPA Contract Period:	1 July 1977 - 31 March 1982
ARPA Contract Number:	DSS MDA-903-77-C-0211
ARPA Order Number:	3396
Program Code Number:	7P10

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SECURE RELIABLE PROCESSING SYSTEMS: FINAL TECHNICAL REPORT, JULY 77 - MARCH 82		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 7/1/77 - 3/31/82
		6. PERFORMING ORG. REPORT NUMBER UCLA-CSD-840228
7. AUTHOR(s) Gerald J. Popek, Principal Investigator		8. CONTRACT OR GRANT NUMBER(s) DSS MDA-903-77-C-0211
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department; School of Engineering and Applied Science; Univ. of California at Los Angeles, Calif. 90024		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 3396
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE February 21, 1984
		13. NUMBER OF PAGES 157
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research 1030 East Green Street Pasadena, CA 91101		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release
Distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

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

This final technical report covers research carried out by the Secure Reliable Processing Systems Contract at UCLA under ARPA Contract DSS MDA-903-77-C-0211 covering the period 1 July 1977 through 31 March 1982. Both computer security work, as well as advances in distributed computing architectures, are discussed.

Secure Reliable Processing Systems
Advanced Research Projects Agency
Final Technical Report
DSS Contract MDA-903-77-C-0211
1 July 1977 - 31 March 1982

1. Introduction

This technical report covers research carried out by the Secure Distributed Processing Systems group at UCLA under ARPA sponsorship during the approximately five year period 1977-1982.

The research conducted during that period focussed on two broad areas: computer security and distributed computer systems. The security work primarily made advances in networks, operating systems, and data management. The distributed systems research was oriented toward systems and software structures suitable for computers connected by local area networks. That work both developed the essential principle of *network transparency,* which largely eliminates machine boundaries from concern to programs and people, and produced an early prototype of a distributed operating system to demonstrate the feasibility of this principle and other architectural approaches.

In the next section, the work done in each of these areas is summarized. Then two sections of substantive technical discussions follow, in the form of technical reports in the primary research areas of security and distributed systems.

2. Computer Security

This work focussed on networks, operating systems, and data management.

2.1. Network Security

In the area of networks, several contributions were made. First, UCLA participated in the larger ARPA sponsored network security experiment employing BCR units to demonstrate that end to end encryption of individual connections on the ARPANET is feasible. UCLA's initial role as a server host in the BCR experiment, and subsequently potentially as a key distribution center, was coordinated with other ARPA contractors.

A major portion of the effort in network security was devoted to the integration of encryption techniques into the protocols of networks and the architecture of the operating systems which are connected. It was found feasible to extend the end of encrypted channels right to the process boundary in host systems, making network control software, as well as all other system software, irrelevant to system security, so long as an appropriate operating system kernel was installed in participating hosts. This development dramatically simplifies the structure of the network security mechanisms, obviating the need for BCR units at secure hosts, as well as any requirement for trusted network management software. A prototype of this integrated end to end network security architecture was developed for the UCLA Secure Operating System Prototype.

The importance of this work is severalfold. First, it demonstrates that end to end security to the process level is very cheap to implement and operate, given the existence of secure operating systems. Second, the approach is directly applicable to existing networks such as the ARPANET.

Next, we performed a general study of encryption and computer networks [POPE79b]. A detailed comparison and evaluation of public key and conventional encryption algorithms was performed from the viewpoint of integration into software systems. Despite the fact that public key systems have additional features beyond those exhibited by conventional encryption, it was found that those features, at least for the functions currently desired in a secure network, are not of significant practical use once the mechanism is integrated into the systems within which it must operate. Those results were reported in the professional literature, as well as in a doctoral dissertation.

The most significant expected advantage of public key encryption over conventional systems concerned digital signatures. Public key systems seemed superior. We have shown that this advantage is largely illusory, because of a defect in all previous signature proposals that permit authors to disclaim authorship at any time. In particular, the problem occurs if the author has access to the information on which the validity of signatures is based. By releasing this information, the author effectively can claim that anyone could forge signatures, including those previously signed.

Once this problem is realized, it becomes immediately clear that some trusted mechanism is necessary to protect the information on which the validity of signatures is based. We call that trusted mechanism the network signature registry. Various designs are possible, especially when such considerations as network, host, and registry failures are considered.

During the current contract period, we have examined these issues in detail, and explored several registry models and digital signature protocols, ranging from fully centralized to fully distributed operation. Both public-key based and conventional based encryption methods have been considered in the solutions. Not surprisingly, here again the particular encryption method seems to make little difference in the final solution.

We developed what appears to be a superior digital signature system. It is based on conventional encryption and a small amount of trusted hardware and software, a signature kernel. During the current contract period, a prototype implementation of the digital signature protocol was implemented under Unix. Several minor errors in the protocol were uncovered in this process, and the result is a rather convincing demonstration of the viability of digital signatures.

2.2. Operating Systems Security

In the area of operating system security, a great deal of progress was made, centering on the UCLA Data Secure Unix prototype kernel. That system's development was completed, full functionality was demonstrated, and results were reported.

Fine grained protection on an individual file basis was implemented, and virtually all standard Unix programs operated without change or recompilation. This system was the first existing and functioning kernel based operating system. It demonstrated the feasibility of this approach.

The UCLA kernel also served as the focus for considerable program verification activity. Complete concrete specifications for the entire kernel, suitable as input to an interactive verification system, were completed. Complete abstract specifications were completed early in 1978, and the abstract to concrete correspondence was also done. The programs being verified composed the largest practical verification effort in the country, and uncovered a great deal about how verification of large systems must be done, as well as the nature of the tools which are required. This research progressed via a cooperative arrangement with USC/Information Science Institute, since their XIVUS interactive verification system was employed in the proofs. The general approach to the verification of large systems is described in [Kemmerer]

2.3. Data Management Security

Data Management Security represented the third focus of the security research effort. Data management systems typically employ considerably more software mechanism in the representation and management of the data they contain than do operating systems. As a result, the task of developing reliable enforcement controls potentially is significantly more difficult. Many have thought as a result that a kernel architecture approach to data management security was not feasible. If so, that would be quite unfortunate, since kernels limit the amount of software which must operate securely. At UCLA, we succeeded in developing a general kernel based dbms architecutre, meaning that it is potentially feasible to provide highly reliable security enforcement in data management systems through the correct installation of a very small amount of software. This result is very important, since without it, much of the code in a data management system would have to operate securely, and the cost of providing secure data management would then often be prohibitive. The design was published late in 1977 and in order to demonstrate its operational feasibility, the INGRES data management system was altered to include the proposed kernel structures. An important result of this test implementation, besides demonstrating feasibility, is that the approach is retrofittable to existing systems. The savings in existing software can be enormous.

2.4. Security Research Comments

This discussion merely summarizes in a brief manner an enormous amount of computer security research conducted during the contract's lifetime. Those interested in a deeper view are referred, first to the technical discussions portion of this final technical report, and subsequently to the appropriate references given at the end of this document.

3. Distributed Computer Systems

The research in distributed computer systems included major efforts to formulate necessary design principles, and then to test and refine those principles via a major implementation of a large scale distributed computing system at UCLA, called LOCUS. The major issues faced in this overall effort, namely transparency, reliability and recovery, and performance, are briefly outlined below, as well as their application to the LOCUS prototype.

The LOCUS system prototype was initially developed as extensive modifications to the Unix operating system on DEC PDP-11s. Subsequently, the same work was done for DEC VAXes. By the end of this contract, development work had progressed far enough that the validity of relevant design principles had been clearly and forcefully demonstrated. However, a great deal remained to be done before a system of value to a substantial user community would result.

3.1. Network Transparency

As real distributed systems come into existence, an unpleasant truth has been learned; the development of software for distributed systems, and for true distributed applications, is often far harder to design, implement, debug, and maintain than the analogous software written for a centralized system. The reasons include a much richer set of error and failure modes to deal with, as well as the lack of a consistent interface across machines by which both local and remote resources can be accessed. An example of a richer error mode is the reality of partial failures: portions of a distributed computation may fail while the others continue unaware. In a centralized system, one typically assumes that the entire computation stops.

Further, local storage may be limited, necessitation that the user explicitly move copies of files around the network, archiving and garbage collecting his own storage. Redundant copies for the sake of reliability is the user's concern. Keeping track of different versions of what is intended to be the same file requires user attention, especially when the copies have resulted from network partitions (leading to parallel changes). As a result, the application program and user must explicitly deal with each of these facts, at considerable cost in additional software. On a centralized machine, with a single integrated file system, many of these problems do not exist, or are more gracefully handled.

An appealing solution to this increasingly serious problem is to develop a network operating system that supports a high degree of *network transparency* ; all resources are accessed in the same manner independent of their location. If *open (file-name)* is used to access local files, it also is used to access remote files. That is,the network virtually become "invisible", in a similar manner to the way that virtual memory hides secondary store. Of course, one still needs some way to control resource location for optimization purposes, but that control should be separated from the syntax and semantics of the system calls used to *access* the resources. Ideally then, one would like the graceful behavior of an integrated storage system for the entire network while still retaining the many advantages of the distributed system architecture. The existence of the network should not concern the user or application programs in the way that resources are accessed.

There are also some system aspects that militate against full transparency. If the hardware bases of each site aren't the same, then it may be necessary to have different load modules correspond to a given name, so that when a user (or another program) issues a standard name, the appropriate file gets invoked as a function of the machine on which the operation is to be performed. There are other examples as well; they all have the characteristic that a standard name is desired for a function or object that is replicated at some or all sites; and a reference to that name needs to be mapped to the local, or nearest instance in normal circumstances.

Nevertheless, in other circumstances, a globally unique name for each instance is also necessary, to install software, do system maintenance functions, etc. A solution that preserves network transparency and provides globally unique names within the normal name space while still supporting site dependent mapping for these special cases is needed. A general *hidden directory* solution was designed to solve these classes of problems.

In summary, Locus contains a general yet high performance solution for transparency in a distributed environment.

3.2. Reliability

Reliability and availability represent a whole other aspect of distributed systems which has had considerable impact on LOCUS. Four major classes of steps have been taken to achieve the potential for very high reliability and availability present in distributed systems with redundancy.

First, an important aspect of reliable operation is the ability to substitute alternate versions of the resources when the original is found to be flawed. In order to make the act of substitution as straightforward as possible, it is desirable for the *interfaces* to the resource versions to look identical to the user of those resources. While this observation may seem obvious, especially at the hardware level, it also applies at various levels in software, and is another powerful justification for network transparency. In LOCUS, copies of a file can be substituted for one another with no visibility to application code.

From another point of view, once a significant degree of network transparency is present, one has the *opportunity* to enhance system reliability by substituting software as well as hardware resources when errors are detected. In LOCUS, considerable advantage is taken of this approach. Since the file system supports automatic replication of files transparently to application code, it is possible for graceful operation in the face of network partition to take place. If the resources for an operation are available in a given partition, then the operation may proceed, even if some of those are data resources replicated in other partitions. A partition merge procedure detects any inconsistencies which may result from this philosophy, and for those objects whose semantics the system understands (like file directories, mailboxes, and the like), automatic reconciliation is done.

Second, the concept of *atomicity* is supported in LOCUS. For a given file, one can be assured that either all the updates are done, or none of them are done. Such a *commit* normally occurs automatically when a file is closed if the file had been open for write, but application software may request commits at any time during the period when a file is open.

Third, even though a high level of network transparency is present in the syntax and semantics of the system interface, each site is still largely autonomous. For example, when a site is disconnected from the network, it can still go forward with work local to it.

Fourth, the interaction among machines is strongly stylized to promote "arms length" cooperation. The nature of the low level interfaces and protocols among the machines permits each machine to perform a fair amount of defensive consistency checking of system information. As much as feasible, maintenance of internal consistency at any given site does not depend on the correct behavior of other sites. (There are, of course, limits to how well this goal can be achieved.) Each site is master of its own resources, so it can prevent flooding from the network.

3.3. Recovery

Given that a data resource can be replicated, a policy issue arises. When the network is partitioned and a copy of that resource is found in more than one partition, can the resource be modified in the various partitions? It was felt important for the sake of availability that such updates be permitted. Of course, this freedom can easily lead to consistency conflicts when the partitions are merged. However, our view is that such conflicts are likely to be rare, since actual sharing in computer utilities is known to be relatively low.

Further, we developed a simple, elegant algorithm to detect conflicts if they have occurred [PARK81]. The core of the method is to keep a *version vector* with each copy of the data object. There are as many elements in the vector as there are sites storing the object. Whenever an update is made to a copy of the object at a given site, that site's element of the version vector associated with the updated copy is incremented. The conflict detection criterion is then very simple. When merging two copies of an object, compare their version vectors. If one dominates the other (i.e. each element is pairwise greater than or equal to its corresponding element), then there is no conflict; the copy associated with the dominating vector should propagate. Otherwise a conflict exists.

Most significant, for those data items whose update and use semantics are simple and well understood, it may be quite possible to reconcile the conflicting versions automatically, in a manner that does not have a "domino effect"; i.e. such a reconciliation does not require any actions to data items that were updated during partitioned operation as a function of the item(s) being reconciled [FAIS81].

Good examples of data types whose operations permit automatic reconciliation are file directories and user mailboxes. The operations which apply to these data types are basically simple: *add* and *remove*. The reconciled version is the union of the several versions, less those items which have been removed. There are of course situations where the system does not understand the semantics of the object in conflict. The LOCUS philosophy is to report the conflict to the next level of software, in case resolution can be done there. An example of this case might be data management software. Eventually, the conflict is reported to the user.

3.4. Performance and its Impact on Software Architecture

"In software, virtually anything is possible; however, few things are feasible." [Cheatham] While the goals outlined in the preceding sections may be attainable in principle, the more difficult goal is to meet all of the above while still maintaining good performance within the framwork of a well structures system without a great deal of code. A considerable amo;unt of the LOCUS design was tempered by the desire to maintain high performance. Perhaps the most significant design decisions in this respect are:

1. specialized "problem oriented protocols",
2. integrated rather than partitioned function, and
3. special handling for local operation.

Below we discuss each of these in turn.

3.4.1. Problem Oriented Protocols

It is often argued that network software should be structured into a number of *layers*, each one implementing a protocol or function using the characteristics of the lower layer. In this way the difficulties of building complex network software are eased; each layer hides more and more of the network complexities and provides additional function. Thus layers of "abstract networks" are constructed. More recently, however, it has been observed that layers of protocol generally lead to layers of performance cost. In the case of up to 5,000 instructions being executed to move a small collection of data from one user program out to the network.

In a local network, we argue that the approach of layered protocols is frequently wrong, at least as it has been applied in long haul nets. Functionally, the various layers were typically dealing with issued such as error handling, congestion, flow control, name management, etc. In our case, these functions are not very useful, especially given that they have significant cost. By careful design, one can build special case solutions which integrate these issues with the higher level functions they support.

These observations lead us to develop specialized *problem oriented protocols* for the problem at hand. In LOCUS, for example, when a user wishes to read a page of a file, the *only* message that is sent from the using site to the storage site is a *read* message request. A *read* is one of the primitive, lowest level message types. There is no connection management, no acknowledgement overheard, etc. The only software ack in this case is the delivery of the requested page.

Our experience with these lean, problem oriented protocols has been excellent. The effect on system performance has been dramatic, as pointed out below.

3.4.2. Functional Partitioning

It has become common in some local network developments to rely heavily on the idea of "servers", where a particular machine is given a single role, such as file storage, name lookup, authentication or computation. We call this approach the *server model* of distributed systems. Thus one speaks of "file servers", "authentication servers", etc. However, to follow this approach purely is inadvisable, for several reasons. First, it means that the reliability/availability of an operation which depends on multiple servers is the product of the reliability of all the machines and network links involved. The server design insures that,for many operations of interest, there will be a number of machines whose involvement is essential.

Second, because certain operations involve multiple servers, it is necessary for multiple machine boundaries to be crossed in the midst of performing the operation. Even though the cost of network use has been minimized in LOCUS as discussed above, it is still far from free; the cost of a remote procedure call or message is still far greater than a local procedure call. One wants a design where there is freedom to configure functions; to avoid serious performance costs, a local *cache* of information otherwise supplied by a server is usually provided for at least some server functions. The common example is file storage. Even though there may be several file servers on the network, each machine typically has its own local file system. It would be desirable to avoid these additional implementations if possible.

An alternative to the server model is to design each machine's software as a complete facility, with a general file system, name interpretation mechanism, etc. Each machine in the local network would run the same software, so system would be highly configurable, so that adaptation to the nature of the supporting hardware as well as the characteristics of use could be made. We call this view the *integrated model* of distributed systems. LOCUS takes this approach.

3.4.3. Local Operation

The site at which the file access is made, (the Using Site, US), may or may not be the same as the site where the file is stored (the Storage Site, SS) or where file synchronization is performed (the Current Synchronization Site, CSS). In fact, any combination of these roles are possible or all may be played by different sites. When multiple roles are being played by a single site, it is important to avoid much of the mechanism needed to support full, distributed operation. That is, when operating essentially locally, performance costs should not increase because of the mechanisms for the general case.

These optimizations are supported in LOCUS. For example, if CSS = SS = US for a given file *open*, then this fact is detected immediately and virtually all the network support overhead is avoided. The cost of this approach is some additional complexity in protocols and system nucleus code.

The system design is intended to support machines of heterogeneous power interacting in an efficient way; large mainframes and small personal computers sharing a replicated file system, for example. Therefore, when a file is updated, it is not desirable for the requesting site to wait until copies of the update have been propagated to all sites storing copies of the file, even if a commit operation is desired. The design choice made in LOCUS is for updated pages to be posted, as they are produced, at the storage site providing the service. When the file is closed, the disk image at the storage site is updated and the using site program now continues. Copies of the file are propagated to all other storage sites for that file in parallel.

4. General Summary

Substantial progress was made by this contract in understanding solutions to fundamental problems in computer security, as well as in distributed computing.

The computer security contributions were as follows. The kernel based approach to operating systems and data management was developed as a concept and implemented to demonstrate feasibility. Advanced program proof techniques were developed and successfully applied to the operating system kernel. Network security problems were addressed by several advances. The ability to add end to end security protocols in a kernel based manner to existing systems was demonstrated. A family of protocols for digital signatures was developed which showed essential functional relationships between conventional and public key encryption.

The distributed system research centered around system design principles and their possible implementation in LOCUS, a distributed version of the Unix operating system. The concept of transparency was developed and demonstrated. Its value was shown to be very high. Methods to support reliability and availability, including version vectors, support for consistent, replicated storage, and a design that permits independent operation in the face of failures, was developed. Methods to reconcile changes made during partitioned operation were also developed. Much of these concepts were demonstrated in one form or another in the LOCUS system prototype.

It is believed that these contributions to computer research represent important and fundamental progress.

5. *Technical Discussion: Computer Security*

Three reports are included in this section. The first discusses proof of security of an operating system; the second describes the use of software architectural methods to construct a secure database system, and the third analyzes the use of encryption in computer networks.

Operating Systems

R. Stockton Gaines Editor

Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek
University of California, Los Angeles

Data Secure Unix, a kernel structured operating system, was constructed as part of an ongoing effort at UCLA to develop procedures by which operating systems can be produced and shown secure. Program verification methods were extensively applied as a constructive means of demonstrating security enforcement.

Here we report the specification and verification experience in producing a secure operating system. The work represents a significant attempt to verify a large-scale, production level software system, including all aspects from initial specification to verification of implemented code.

Key Words and Phrases: verification, security, operating systems, protection, programming methodology, ALPHARD, formal specifications, Unix, security kernel

CR Categories: 4.29, 4.35, 6.35

† Unix is a Trademark of Bell Laboratories.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211. Authors' present addresses: B.J. Walker and G.J. Popek, Department of Computer Science, University of California, Los Angeles, CA 90024; R.A. Kemmerer, Computer Science Department, University of California, Santa Barbara, CA 93106
© 1980 ACM 0001-0782/80/0200-0118 $00.75.

1. Introduction

Early attempts to make operating systems secure merely found and fixed flaws in existing systems. As these efforts failed, it became clear that piecemeal alterations were unlikely ever to succeed [20]. A more systematic method was required, presumably one that controlled the system's design and implementation. Then secure operation could be demonstrated in a stronger sense than an ingenuous claim that the last bug had been eliminated, particularly since production systems are rarely static, and errors easily introduced.

Our research seeks to develop means by which an operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components of this task are: (1) developing system architectures that minimize the amount and complexity of software involved in both protection decisions and enforcement, by isolating them into *kernel* modules; and (2) applying extensive verification methods to that kernel software in order to prove that our of *data security* criterion is met. This paper reports on the latter part, the verification experience. Those interested in architectural issues should see [23]. Related work includes the PSOS operating system project at SRI [25] which uses the hierarchical design methodology described by Robinson and Levitt in [26], and efforts to prove communications software at the University of Texas [31].

Every verification step, from the development of top-level specifications to machine-aided proof of the Pascal code, was carried out. Although these steps were not completed for all portions of the kernel, most of the job was done for much of the kernel. The remainder is clearly more of the same. We therefore consider the project essentially complete. In this paper, as each verification step is discussed, an estimate of the completed portion of that step is given, together with an indication of the amount of work required for completion. One should realize that it is essential to carry the verification process through the steps of actual code-level proofs because most security flaws in real systems are found at this level [20]. Security flaws were found in our system during verification, despite the fact that the implementation was written carefully and tested extensively. An example of one detected loophole is explained in §2.5.

This work is aimed at several audiences: the software engineering and program verification communities, since this case study comprises one of the largest realistic program proving efforts to date; the operating systems community because the effort has involved new operating system architectures; and the security community because the research is directed at the proof of secure operation. We assume the reader is acquainted with common operating system concepts, with general program verification methods, and with common notions of abstract types and structured software. Understanding of Alphard proof methods [33], upon which this work is based, is not necessary since relevant aspects are summarized here.

1.1 System Architecture

The Data Secure Unix (DSU) architecture supports a full function multiprogramming, uni-processor operating system running on a DEC PDP-11/45 computer, with application code interface essentially identical to standard Unix [24]. In our system the software relevant to protection is isolated into a few components. This paper is concerned with the kernel, a component which depends only on the semantics of the hardware, thereby providing the base for the entire system. The kernel supports processes, capabilities, pages and devices via kernel calls. Examples of kernel calls to manipulate these objects are: initialize-process, invoke-process, grant-capability, swap-in-page, swap-out-page, memory-management-interrupt, send-interrupt, set-interrupt, hardware-interrupt, and start-io. Part of supporting these objects involves enforcing an access policy, thereby protecting the objects. The kernel does not determine the protection policy it enforces, however; that task is performed by a second level kernel called the *policy manager*[1]. The policy manager, which runs as a process, informs the kernel of the security policy, i.e. the rules which state what any user is entitled to access, by issuing the grant-capability kernel call.

The two-level kernel design splits the task of proving that the system is secure into two subtasks: a) verifying that the kernel enforces the security controls set by the policy manager and maintains the integrity of the object types it supports, and b) verifying that the policy manager sets the kernel controls to implement the desired policy properly. There are several advantages to splitting the security-relevant code in this fashion. First, splitting a verification task in general decreases the total effort involved. Second, different policies can easily be implemented. Third, verifying that the implemented policy reflects a given abstract policy is not encumbered by the relatively complex task of enforcing that policy. Most importantly, however, the abstract specifications for the enforcement portion can be expressed straightforwardly and subsequently proven, independent of policy issues [11]. This simplification is significant since many of the successful security penetrations of systems have occurred because of errors in the enforcement rather than in the policy part of the system.

Both the kernel and the policy manager are implemented as sequential software, although in different ways. Thus the specification and verification tasks can avoid the difficulties of parallelism. Each call to the kernel runs physically uninterrupted, and thus cannot wait for an input/output completion since this would result in an intolerable delay. Of course, a call can *start* an I/O operation for a process. In addition, since I/O is the only source of parallelism once interrupts are disabled, it is necessary to insure that no I/O affects any internal kernel data or kernel-call parameters. This is accomplished by copying kernel-call parameters into kernel data space before kernel-call processing and by coding the kernel I/O routines to avoid I/O to or from the kernel address space. The policy manager process does not run physically uninterrupted but is nevertheless logically sequential. Interrupts and messages are queued for the process to inspect; the process contains no asynchronous jumps.

The other important parts of the DSU system are the dialoger, scheduler and application processes. The dialoger handles user authentication and protection policy modifications. This process is trusted and therefore must be verified. On the other hand, the scheduler, which handles memory and cpu allocation, is not security relevant because while it makes decisions about what process to execute and what pages to locate in core, all the actual sensitive operations are done via kernel calls. Application processes run with two linear address spaces. One address space typically contains kernel-interface software and thereby provides a standard Unix environment for the application code, which runs in the second address space. The correctness of the interface package is not relevant to data security, since a separate logical copy of the package is encapsulated within each process. More detail is given in [21, 23].

1.2 Proof of the Security of the Kernel

The proof that the kernel is secure has two parts: a) developing four levels of specifications, ranging from Pascal code to the top-level security criterion, and b) verifying that the specifications at these different levels of abstraction are consistent with one another. Figure 1 outlines the four levels of specification and the three consistency proofs, giving section numbers where more detail is provided. The top-level formalizes our intuitive notion of data security for the kernel. The abstract level contains operations corresponding to uninterrutible kernel calls and more specific objects like pages, processes, and devices. The low-level specifications, which use data structures from the implementation, are still at a higher level than the code because detail is omitted and some abstracting of data structures is employed.

The four levels of specification are described in §2 while the consistency proofs and mappings are considered in §3.

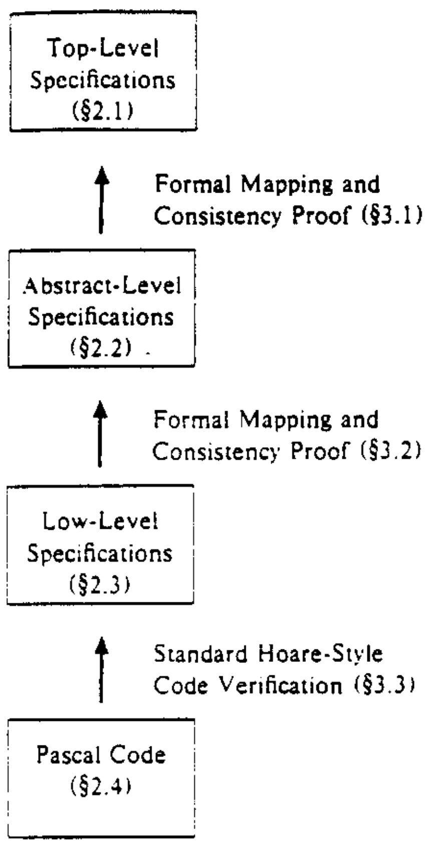
This proof work assumes that the Pascal compiler, interactive verification tools, and hardware operate correctly. We leave these issues to others.

2. Specification

One approach to developing a secure implementation is to work *top down*: that is, develop the top-level, abstract-level, low-level and code, in that order

[1] References to *the* kernel in this paper refer to the base or first-level kernel. The second-level kernel is always referred to as the policy manager.

Fig. 1. Specification levels and consistency proofs.


Top-Level Specifications (§2.1)
Formal Mapping and Consistency Proof (§3.1)
Abstract-Level Specifications (§2.2)
Formal Mapping and Consistency Proof (§3.2)
Low-Level Specifications (§2.3)
Standard Hoare-Style Code Verification (§3.3)
Pascal Code (§2.4)

Although this is a fine approach in principle, it is often not practical. On several occasions we altered the functionality of the system as we better understood the environment and the verification constraints. For example, only after a first design and implementation did we understand detailed I/O issues well enough to develop the improved, verifiable I/O design described in §2.3.2. The design also evolved as a result of demanding efficiency constraints. Thus, even though much effort was spent building verifiable software, an informal design process and most of the coding preceded the major work on formal specifications. We suspect that this experience will not be uncommon[2].

In generating each set of specifications, there were several key considerations:

- the specification language to be used,
- the intent of the specifications and thus their content, and
- the organization of the specifications.

The following subsections consider each set of specifications with these considerations in mind.

[2] A similar conclusion was drawn by Gerhart in her verification effort [2, 3].

2.1 Top-Level Specifications

Clarity and precision are essential in the top-level specifications. Ultimately, people must decide whether the properties specified are the ones that they desire. Consistency of lower levels in principle may be checked by machine.

Our top-level specification is a *security criterion*: a specification of the protection enforcement to be provided by the kernel. This criterion must be expressed in a formal but easily understood and well defined language. We chose a finite state machine model,

$$M = < S, s_0, T >$$

where S is the set of states in which the machine may exist, s_0 is the initial state and T is the transition function. The transition function expresses the effect of a single operation which in turn corresponds to a number of operations at the abstract level. A state of the machine consists of the following uniquely named, non-overlapping components:[3]

a) **process** objects, which at this level of detail have unstructured values,
b) **protection-data** objects (one per process), with values being sets of capabilities,
c) **general** objects, again with unstructured values (pages and devices are considered **general** objects at this level of specification), and
d) a **current-process-name** object, whose value is the name of the process currently running.

The top-level security criterion is as follows: A component of the state is actually modified or referenced only if the **protection-data** for the **process** named in the **current-process-name** object allows such access to the component. Thus, at the top level of specification, we merely care that if a component is modified or referenced then that access is allowed; we do not care how a new value for a component is derived. The formal version of this criterion, which is given elsewhere [29], is largely based on the work reported in [22, 11].

This security criterion is meant to capture the common notion of *data security*, and is free of any particular policy. As pointed out in §1.1, the kernel is responsible only for *enforcement* of whatever policy is implemented in the policy manager.

2.2 Abstract-Level Specifications

Like the top-level model, the abstract-level model is a finite state machine. The transition function at this level expresses the effect of each kernel operation. The abstract-level state consists of sets of objects, examples of which are *Process*, *Device*, *Device-status*, *Page*, *Protection-data* and *Kernel-data*. Most of these objects map easily to objects at the top level. Unlike the unstruc-

[3] The convention in this paper is for top-level components to appear in lower case bold roman, abstract-level components to start with an upper case italic letter followed by lower case italics, and low-level components to be completely lower case italics.

tured value component of objects in the top-level state, however, the value components of the abstract-level objects are declared in some detail. The *Kernel-data* object, for example, has a value component made up of structures necessary to support the top-level and abstract-level object abstractions. *Incore-page-table*, which is involved in the *Page* abstraction, is one such structure. Using the data declaration facilities of Pascal [9, 32], it is:

```
Incore-page-table: array [0 .. Num-pages -1] of
  record
    Pgtbl: Id;  The name (identifier or id) of the page
        resident in this page-frame;
    I/o-cnt: integer;  A counter of the number of
        ongoing I/O's involving this page-frame.
    Clean: boolean;  A flag indicating if the contents of
        the page in this page-frame differ from the
        contents of the secondary storage copy of the
        page.
    Swapping: boolean;  A flag set while the I/O to
        bring the page into core is in progress.
  end;
```

Appendix C uses this data structure in the low-level to abstract-level mapping.

Unlike the top-level abstraction, the abstract-level has considerable detail, needed primarily for the security proofs of trusted processes [4], in our case the policy manager. Verification of trusted processes, like all verification, requires the complete semantics of all the primitive operations, in this case kernel calls, which are executed.

The abstract model is complete, in that every aspect of the kernel which can be sensed from non-kernel code is included, even details which are used only by the policy manager and scheduler, and are not visible to user software[5]. Since a general user need not be aware of all of the kernel structures and operations, two views of the abstract model can be given: the user view and the special process view. Most of the abstract model's details are relevant to only the policy-manager and the scheduler processes, so the user's view of the abstract model is quite simple.

A detailed set of abstract-level kernel specifications also provides a framework for analysis of interprocess confinement channels [13]. Without complete specifications, one could design a kernel that meets our top-level security criterion, yet contains kernel calls which permit one process to store information in a kernel table and another process to sense the values in that table. While we do not formally consider confinement control, our detailed abstract-level specifications do provide a foundation for bandwidth analysis.

While the abstract-level specifications contain more detail than one might have expected, there is still considerably less detail than at the low level. Redundant data structures at the code level, introduced to maintain acceptable performance, are not part of the abstract-level state. For example, neither the process-index field of a process's argument-block, which allows the kernel to determine the owner of the argument-block without extensive searching, nor the page-frame-index field associated with each relocation register, which indicates the page-frame to which the relocation register is mapped (also available by analysis of the memory address part of the relocation register) is reflected in the abstract model. The abstract level also often presents a single data structure where the implementation uses several (see the mapping function in appendix C). Therefore, while the formal abstract description is more detailed than one might expect or prefer, it is still considerably easier to understand than the low-level system it represents.

2.3 Low-Level Specifications

The low-level specification effort uses a finite state machine model, just as the top and abstract-levels did. Each kernel call represents a transition, or more precisely, an equivalence class of state transitions, each member of which is the call with some set of parameters. The low-level state consists of those global kernel variables that retain values from call to call. Some additional abstract structures are needed, however. Section 2.3.2 describes, for example, how the low-level state can contain a table which resides on disk.

Deciding what to specify about the states and the transitions was straightforward: all modifications of the state components have to be accounted for in the specifications. The specification language was determined by the XIVUS system [4, 34] under development at Information Sciences Institute (ISI). The specification language required by XIVUS is a superset of first order predicate calculus, with additional convenience coming principally from constructs for expressing updated values of arrays and records.

The task of organizing the low-level specifications received a great deal of attention, and resulted in an organization technique referred to here as *data-defined specifications*. The next section describes this technique while the following section describes special issues in the specification of hardware semantics.

2.3.1 A Data-Defined Specification Technique.

In large software systems, the size of the set of formal specifications is quite large. These specifications must interface with both abstract-level specifications and code. Consequently, the organization of the low-level specifications is critical. The data-defined specification technique draws from both the techniques of abstract data

[4] We believe that any kernel-based system will contain trusted processes. See §4.

[5] In the implementation of the kernel the capabilities necessary to successfully execute some of the operations (eg. grant-capability and invoke-process) are given only to the policy-manager or the scheduler. Therefore, these operations cannot be successfully executed by application processes.

types [15, 16, 27, 33] and functional specifications [19, 26].

A finite state machine can be considered as one large abstract data type, with the machine state and transitions being the data type components and operations. Although this organization is theoretically sufficient, it does not constrain the form of the specifications for each operation. As explained below, the nature of low-level properties and kernel call operations makes this gross level of organization unsatisfactory.

Several characteristics of the low-level state and associated operations significantly affected the organization of the specifications. First, many properties must be true both before and after each kernel call. These properties, which characterize each low-level state, are called invariants [7, 33]. A simple example of an invariant is a range constraint on an array index stored as part of the low-level state. Second, many invariants cross data-structure boundaries. Appendix A describes one such invariant which involves two relocation register structures, the page-table, the capability lists, the running process index and part of the process table. Finally, much of the kernel-call code is directed at maintaining the invariants.

Since code involved in maintaining each invariant will in general be executed in several kernel calls, a purely functional set of specifications, that only specifies the actions of each kernel call and does not use invariants, would contain considerable redundancy. This redundancy presents two problems. First, it complicates the complete set of kernel-call specifications, increasing the chance of specification inconsistency or omission and possibly confusing a human reader. Second, the proof which shows that the low-level specifications imply the abstract-level specifications is expanded unnecessarily. The invariant property would be expressed in a functional form in several kernel calls and thus would be involved in several kernel-call proofs instead of in a single one. Proofs of invariants and individual kernel calls are discussed in §3.

An abstract data type approach that involves more than one large data type might solve the redundancy problems. However, incorporating specifications that cross data-type boundaries is contrary both to the spirit and mechanism of current abstract-data-type frameworks. Structures involved in each invariant would have to be part of the same data type, and looking at the complete set of low-level invariants, one observes that almost all the data structures would have to be combined into one or two types, thereby defeating the advantages of abstract data types[6].

Though it is neither a strict abstract-data-type technique nor a strict call-by-call technique, the data-defined specification technique draws from both. Although specifications refer only to the values of variables before or after kernel calls, the specifications were organized not on a call-by-call basis but on a data-structure basis. In particular, the presentation of the formal set of kernel specifications discusses each data structure in turn, referring to specific kernel calls as little as possible. Unlike an abstract-data-type organization, specifications (eg. invariants) crossing data-structure boundaries are permitted. Each of these specifications is presented inside one of the relevant data-structure's set of specifications and referenced in the other data-structures' sets of specifications.

The discussion so far has mentioned invariants as the only form of specification. Invariants, however, can only be used to specify the properties of a state; they cannot be used to specify the effect of a state transition. If it were possible to define a *secure state*, the invariant form of specification would be sufficient. Unfortunately, it is often the case that the conditions under which a state change can occur must be specified. For example, data security requires that the protection data (i.e. the data representing the policy implemented by the policy manager and enforced by the kernel via capability lists for each process) be modified only by the policy manager in either the grant-capability or initialize-process kernel calls. Other examples of relevant state changes include cases where the modification of one data structure is conditioned on the modification of another data structure, independent of what kernel call made the modification. Each of these forms of specification requires the ability to refer simultaneously to the values of the state both before and after the kernel call. Invariants do not provide this ability, so a different form, called a *conditional*, was introduced. The format of conditionals is suggestive of their name; it is "*condition*1 *implies condition*2". Typically, *condition*1 either lists a particular kernel call (eg. call = grant-capability), or it indicates that some variable(s) changed value. (eg. var ≠ var')[7]. An example of the latter form of conditional is given in Appendix B.

While the data-defined specification technique allows one to present cleanly a complete set of kernel specifications, they are not in the proper form for standard Hoare-style, code-level verification [5]. Entry/exit assertions are needed for each kernel call[8]. Since the data-defined-specification technique organizes the specifications on a data structure basis and the specifications consist of a set of invariants and conditionals, the exit assertion for each call is the conjunction of all the invariants and conditionals. The entry assertion is the conjunction of all the invariants. While these entry/exit assertions are easy to generate, they are overly lengthy.

The number of conjuncts in both the entry and exit assertions can be substantially reduced by the automatable application of some simple logic. To do so, it is necessary to parameterize both the kernel calls and the specifications. Although the Pascal language forces the coder to declare parameters for procedures as either by-

[6] Although abstract data types did not seem practical in general, there were several places where they were used. In particular, §2.3.2 describes how abstraction was used for device registers.

[7] The value of variables before a kernel call are denoted by the name of the variable suffixed with a prime (').

[8] Verifying a piece of code with respect to an entry/exit assertion pair means that if the entry assertion is true before executing the code and the code terminates, the exit assertion will be true at termination.

reference or by-value, global variables cannot be similarly declared. Since the low-level state is made up entirely of global variables, we implemented import lists, a language construct borrowed from Euclid [14]. For each procedure and function, an import list specifies which global variables are used by-value and by-reference in that routine and all routines called by it. In particular, each kernel call has an import list and is consequently parameterized. Invariants and conditionals are also parameterized, the parameter list being the free variables in the specification.

To shorten a kernel call exit assertion, a comparison of the kernel call's by-reference import list and each invariant's parameter list is made. If the lists are disjoint, the invariant trivially holds across the kernel call. Some conditionals can be handled in an analogous way: those conditioned either on the modification of some data structure or the execution of a certain kernel call. In the former case, the conditional is trivially satisfied if the given data structure is not imported by-reference. In the latter case, only the conditional relevant to the kernel call under consideration need be proven explicitly. All other conditionals will have false conditions, implying a true specification. By the use of import lists in this way, a manageable kernel call entry/exit specification can be generated mechanically from the invariants and conditionals. Additional heuristics can be applied for further reductions [28].

An important advantage which results from the data-defined organization of the low-level specifications is that it presents an alternate view of the kernel. That is, for the programmer, the code is a procedurally oriented description of the system, while the data-defined specifications form a somewhat orthogonal view that cuts across all calls.

2.3.2 Hardware Semantics. The semantics of high-level-language programs can typically be defined by the definition of the high-level language in which the programs are written. Unfortunately, that statement is not correct for an operating system nucleus. Hardware characteristics show through, thereby making a language definition such as the axiomatic specification of Pascal [8] incomplete in critical ways.

First, kernel software is responsible for maintaining important aspects of the environment in which that kernel software itself runs. For example, on a PDP-11, privileged kernel mode, in which the kernel runs, operates in a virtual address space. The page registers controlling that space and maintaining the linearity assumed by the compiler are themselves controlled by the kernel code whose validity depends on their correct control.

Second, the finiteness of hardware is often masked incorrectly by the programming language compiler. For example, the available compiler translates Pascal arithmetic straightforwardly into machine code. However, arithmetic on a PDP-11 has 16 bits of precision, despite the fact that addresses are 18 bits, and protection-critical validity calculations using those addresses must be made.

A more serious problem results from the finite capacity of main storage. There is a capability list for each process, and it is not possible for them all to reside in main memory. Each capability list is on a page that can be moved in and out of main memory by swap-in-page and swap-out-page calls issued by the scheduler process. Nevertheless, kernel code references and updates each list. The semantics of the kernel Pascal code has to reflect a complete static data structure containing all the capability lists in kernel storage. The problem is handled by declaring, in kernel code, a single capability list structure. In an isolated routine, kernel page relocation registers are manipulated so that a single capability-list page is included in the kernel address space, at exactly the virtual location expected by the declaration. The semantics for this routine requires an abstract array of capability lists, so that the page register change has the effect of reassigning all the values previously contained in the real capability-list data structure[9].

Last, the I/O interface on the PDP-11 is surprisingly complex. For each physical device, there are up to 24 registers with quite complicated semantics, and the meaning of each field in a register differs substantially among devices. I/O is done by loading values into the registers through normal Pascal assignment. It is quite typical, as a result, to have a substantial amount of device specific software in the nucleus of software systems controlling many such devices. Therefore, in the DSU kernel, a considerable effort was spent to make as much of the code as possible device independent [28]. This effort resulted in substantially reducing the overall length of the code, and left only small device dependent routines, called device drivers, each of which only loads and/or checks the device registers for the different devices. Device independent entry/exit assertions (i.e. semantics) were written for these device drivers to allow the verification of the remaining device independent kernel software. Bit level device register manipulation is hidden, in the normal manner of abstract data types.

The mapping from the the device independent driver semantics to the actual register manipulation involves a clear definition of how the device operates, but is quite straightforward. By successfully making almost all the code device independent, that specification effort is greatly reduced. In addition, adding new devices is not only easy to code but involves little or no additional specification and little additional verification. The importance of this I/O abstraction is hard to overestimate, since approximately half of the kernel is concerned with I/O, even after the simplifications.

The general approach employed to deal with hardware issues isolates them as much as possible, so that most kernel code executes in a standard Pascal environment. Conventional notions of data abstraction are used in this part of the software architecture. Code fragments which

[9] Appendix A illustrates an invariant which uses the abstract capability list.

reference or modify hardware registers and which therefore do not have standard Pascal semantics were put in separate procedures and functions. As a result the exit assertions of these procedures and functions can reflect the unusual semantics in a way that can be used elsewhere in the specifications and proofs, hiding the manner in which the semantics were realized.

2.4 The Pascal Code

Like the other levels of specification, the Pascal code must interface with its surrounding levels, in this case the hardware and the low-level specifications. This section outlines how these interface requirements affect the code.

Pascal was chosen as the programming language primarily because its semantics had been formally defined [8] and XIVUS uses Pascal [34]. We did make some small changes to the language. One change, mentioned earlier (§2.3.1), is the addition of import lists to aid verification. Another addition is the ability to declare variables to be located at specific memory locations. This ability allows hardware register manipulation to be done without assembly language routines. Adding the hardware register manipulation directly into Pascal is not problem free, as we have pointed out, because the semantics of assignment to some registers is not consistent with standard Pascal assignment semantics. Consequently we separate such assignments into individual procedures.

Heavy use of procedures is probably the most distinctive aspect of the flow of control in the kernel. In addition to facilitating human understanding, this practice can reduce the number of proofs required. When code common to several kernel calls is separated into a procedure, it need be verified only once. Also the number of proofs which result from a fragment of code is equal to the number of paths in the code from assertion to assertion, and the number of paths can often be reduced either by introducing intermediate assertions or by breaking the code into procedures. Figure 2 illustrates this point.

Fig. 2. Using intermediate assertions or subprocedures to reduce the number of proofs.

begin
  entry a1;
  if *c*1 **then** *A*
      else *B*;
  if *c*2 **then** *C*
      else *D*;
  if *c*3 **then** *E*
      else *F*;
  exit a2;
end.
	begin
	  entry a1;
	  if *c*1 **then** *A*
	      else *B*;
	  if *c*2 **then** *C*
	      else *D*;
	  assert a3;
	  if *c*3 **then** *E*
	      else *F*;
	  exit a2;
	end.
		begin
		  entry a1;
		  if *c*1 **then** *A* **else** *B*;
		  if *c*2 **then** *C* **else** *D*;
		  call *SUB*;
		  exit a2;
		end.
		proc *SUB* **begin**
		  entry a4;
		  if *c*3 **then** *E* **else** *F*;
		  exit a5;
		end.

There are 8 paths and thus 8 proofs.	There are 4 paths to the assert statement and 2 paths from it; only 6 proofs.	There are 4 paths in the mainline and 2 in the subprocedure; only 6 proofs.

Keeping the code of the kernel short and simple is often not compatible with efficiency. For example, in several places array-index guesses are kept to help reduce search costs. The total code is of course larger since both the long searches and the searches via the guesses must be included. Consequently the addition of the guesses results in a slightly more complicated verification task. The data structures in general, however, are very simple. For several reasons, neither dynamic storage allocation nor pointers are used. First, neither the compiler implemented at UCLA [30] nor the XIVUS verification system handle these constructs. Second, at the time coding first began, it was not known how difficult verification involving these structures would be[10]. Finally, many of the data structures are not dynamic so neither dynamic storage nor pointer usage seemed to have any advantage over arrays. Arrays, records, and most commonly, arrays of records are the only data structures. Although UCLA Pascal has no operations involving entire records, data items were organized into records for two reasons. First, the code and documentation was clearer. Second, import lists were simplified, because entire records could be imported without having to list each element.

Being able to import records only in their entirety turned out to be a mixed blessing. Importing an entire record by-reference when only one or two fields were being modified shortens the import list but it complicates the entry/exit assertions. At the kernel-call level, if a data item is imported by-reference, all the assertions involving that data item must in some way be proven. To avoid having to list and prove assertions about data items which are not modified, assertions to that effect are included. These assertions, which might have to be restated in several subprocedures, can be avoided if the unmodified data items are not part of a record containing data items which are modified. There are in general two ways to reduce the significance of the above problem. One is to change the import list mechanism to permit subrecords to be imported. The more easily implemented solution is to split the records whose fields have different modification characteristics into separate records. As a matter of expediency the latter solution was adopted.

2.5 Reflections on the Specification Effort

Over ninety percent of the UCLA kernel was successfully specified in the manner outlined in the previous sections. This success is generally encouraging, but the

[10] Work subsequently done both at Stanford [17] and UCLA [1] has shown that code employing these structures need be no more difficult to verify than arrays.

effort required is also sobering. The task is still too difficult and expensive for general use.

Nevertheless, our specification work was very valuable, not only as a research effort, but because significant security errors were uncovered. For example, although the code had already been tested and executed properly for the normal case, a boundary condition of the kernel call which maps a page into the user's address space had not been handled properly. The result allowed a mischievous process to read or modify memory pages adjacent to its own. Since protection-data pages are allocated to arbitrary page-frames, this mischievous process could, if patient, modify its protection-data to give itself access to any object [28]. The discovery of flaws of this nature clearly justifies the need for formal specification, if not complete verification, for security relevant and other sensitive applications.

During the specification effort, we were also surprised by the number of apparently intrinsic relationships among the kernel data structures. This pattern is in part because many of the separable functions had already been removed from the kernel. However we believe that, as verification techniques are more widely applied, such relationships will be encountered more frequently.

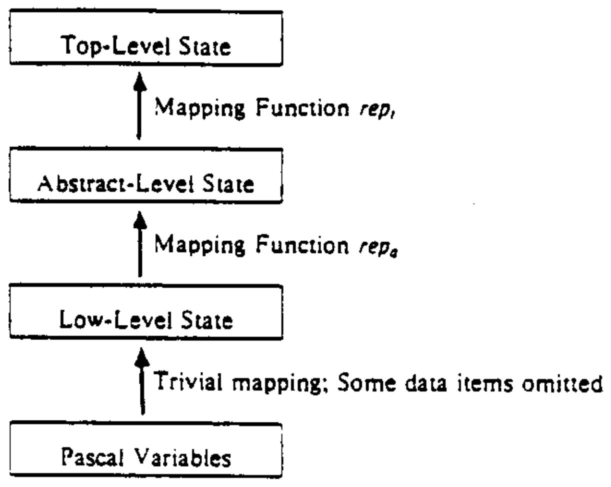
In general, we must conclude that more aid is needed, and research justified, to ease the specification task. Useful areas to pursue include: a) programming language design to help capture and enforce some of the properties for which we were forced to use general verification methods to assure, b) specification language design and c) techniques for determining completeness and consistency of the specifications, particularly with respect to hardware.

3. Proofs

All of the specifications at all of the levels, with the exception of the code, are written in predicate calculus. However, since the data structures are different at different abstraction levels, a mapping function from each specification level to the next higher level is used to construct higher-level data structures out of lower-level ones. Such a mapping can be used to show that two levels of specification have the same meaning. We have used several levels of specification and mapping to verify that the top-level specifications are consistently implemented by the Pascal code for each kernel call. Figure 3 illustrates the states at the various levels, together with appropriate mapping functions. Appendix C contains an example of one of the mapping functions, rep_a.

The verification techniques of Alphard have been adapted as a basis for doing the mappings and consistency proofs. The Alphard methodology provides a formal framework to prove that abstract specifications of an abstract data type are consistent with the code that implements that type [33]. The parts of the proof, which must be done for all values of x, the value of the low-level

Fig. 3. States and the associated mapping functions.



state, are described below, both informally and formally[11].

(1) Establish the validity of the low-level representation — any combination of low-level variables that satisfy the low-level invariant corresponds to an abstract object that satisfies the abstract invariant.

$Linv(x) \rightarrow Ainv(rep_a(x))$

(2) Establish the low-level invariant and certain initial properties of the abstract state after executing the initialization code.

$true\ \{OP_{init}\}\ (Linv(x) \wedge Ainit(rep_a(x)))$

(3) For each operation j, prove the code to be consistent with the low-level entry/exit or pre/post assertions, including the low-level invariant.

$\forall j, ((Lpre_j(x) \wedge Linv(x))\{OP_j\}(Lpost_j(x) \wedge Linv(x)))$

(4a) For each operation j, show that if the abstract operation could legally be applied (i.e. $Apre_j$ is true) and the low-level invariant is true, then the low-level entry assertion is true. As explained below, this part of the proof is trivially done for the UCLA application.

$\forall j, ((Apre_j(rep_a(x)) \wedge Linv(x)) \rightarrow Lpre_j(x))$

(4b) Prove that if the low-level operation is properly invoked, then the post condition of the low-level operation is sufficient to imply the post condition at

[11] The formalism uses simple first order predicate calculus and standard verification notation. The first order predicate calculus notation is implication ($\rightarrow$), conjunction ($\wedge$) and universal quantification ($\forall$). The verification notation is "a {$prog$} b", where a is the entry or pre condition and b is the exit or post condition and a proof means that if a is true before executing $prog$, then b is true afterward, given that $prog$ terminates.

the abstract level. Here x' means the value of x before invoking the low-level operation.

$$\forall j,x',\ ((Apre_j(rep_a(x')) \wedge Linv(x') \wedge Lpost_j(x)) \rightarrow Apost_j(rep_a(x)))$$

Our application fits fairly well within the Alphard framework. The operations are just kernel calls. The initialization code is the kernel initialization. The kernel was designed to allow any sequence of kernel calls; calls fail if they are improperly invoked but they can always be executed. Thus, there are no preconditions on the operations. Consequently part 4a of the above proof is trivially true. Also parts 3 and 4b can exclude references to $Lpre_j$ and $Apre_j$.

Additional work, however, results from the addition of a top level above the abstract level. To do the proof between the abstract and top levels requires repeating parts 1 and 4b of the Alphard proof components. The additional part 1 becomes:

$$(1')\quad Ainv(rep_a(x)) \rightarrow Tinv(rep_t(rep_a(x)))$$

The additional part 4b is:

$$(4b')\quad \forall j,x'.\ (Ainv(rep_a(x')) \wedge Apost_{j(x)}) \rightarrow Tpost(rep_t(rep_a(x)))$$

Lastly, part 2 is modified to combine initialization requirements at each level of abstraction:

$$(2')\quad true\ \{OP_{init}\}\ (Linv(x) \wedge Ainit(rep_a(x)) \wedge Tinit(rep_t(rep_a(x))))$$

Section 3.1 discusses the abstract to top level proof (parts 1' and 4b'); section 3.2 discusses the low-level to abstract proof (parts 1 and 4b); and finally, section 3.3 considers verification of the actual code with respect to the low-level specifications (parts 2 and 3).

3.1 Top-Level and Abstract-Level Correspondence

These proofs are in most respects straightforward, so we simply outline them here. Since the unique top-level operation is implemented by many operations at the abstract level, it is necessary to show that each of these operations consistently implements the top-level operation. To prove that the abstract level is a consistent implementation of the top-level specification it is necessary to show that if the abstract invariant holds for the abstract level components, then the top-level invariant holds for the top-level components. That is,

$$(1')\qquad Ainv(rep_a(x)) \rightarrow Tinv(rep_t(rep_a(x))).$$

The top-level invariants require the components be disjoint and have unique names. These proofs are straightforward since the name-mapping function is an identity function and the top-level components are constructed directly from unique components at the abstract level.

In addition to showing that the top-level invariant holds whenever the abstract-level invariant does, the following condition must be shown for each of the abstract operations: If both the exit assertion for an operation and the abstract-level invariant hold, then the exit assertion for the top-level operation is satisfied (part 4b' of the proof). To prove each of these theorems, we first consider the modification operation, then the more difficult case of reference. Assume that a top-level component (O_t) is modified[12]. If the exit assertion for the abstract-level operation specifies that the abstract-level component (O_a) which represents the top-level component is unchanged, we have a contradiction. That is, since $O_t = rep_t\ (O_a)$, if $O_a = O_a'$ this implies that $rep_t\ (O_a) = rep_t\ (O_a')$, and thus $O_t = O_t'$. Therefore, we can conclude that our assumption (i.e., that O_t is modified) is false and the proof is complete for this particular component. However, if it is not specified that $O_a = O_a'$ in the abstract-level specification, then the proof can be only completed if the abstract-level exit assertion specifies that some type of modify access (init, grant, write) for O_a be contained in the calling process's *Protection-data* object. Since the calling process's *Protection-data* object at the abstract level is mapped to the calling process's **protection-data** object at the top level and modify access at the abstract level is mapped to modify access at the top level, the proof is complete for this top-level component. Similar proofs must be completed for each of the top-level components.

The proofs for the case of reference are performed similarly, except that a reference at the top level can be represented in many ways at the abstract level. Therefore, determining whether an abstract level object is referenced is much more difficult than determining whether or not one is modified. This problem is discussed in [11]. Once it is established whether or not O_a is referenced the proof proceeds in the same manner as for the modify case. That is, if O_a is not referenced we have a contradiction, and if O_a is referenced it is necessary to prove that the abstract-level *Protection-data* object for the calling process contains an element that allows reference type access (eg., read) to O_a.

This portion of the verification task is essentially complete. Details may be found in [29].

3.2 Low-Level and Abstract-Level Correspondence

This section discusses the theorems that relate the low-level and abstract-level invariants, and the low-level and abstract-level exit assertions. Because these theorems are much more complex than the abstract to top-level theorems, a proof framework was selected that permits a reader to look at any step of a proof and, based on the annotation, determine how one arrives at the statement given for that step. The proofs were done by hand, using a derivation approach presented by Kalish and Montague [10], and then the AFFIRM verification system [18] at ISI was employed as a proof checker to verify the

[12] For our discussion in this paper we assume that if O_t is modified, then $O_t \neq O_t'$. Some definitions of modify (e.g., assignment of a value) allow the value of a modified component to be the same as the value it had before the modification took place.

proofs[13]. We do not suggest that this method is feasible for all large scale verification efforts, but only demonstrate that it is possible. The method was chosen because when the theorem prover is used in an unguided way to *prove* a theorem, the resulting proofs are often difficult to follow or decipher. The prover sometimes arrives at a result through manipulations that are not obvious.

The theorem relating the low-level and abstract-level invariants states that if the low-level invariant holds at the low level, then the abstract invariant holds at the abstract level (i.e., part 1: $Linv(x) \rightarrow Ainv(rep_a(x)))$. Since both *Linv* and *Ainv* are composed of the conjunction of a number of assertions, the theorem can be expressed as:

$(h_1 \,\&\, h_2 \,\&\, h_3 ...) \rightarrow (c_1 \,\&\, c_2 \,\&\, c_3 ...)$

where the h_m's are the hypotheses and the c_n's are the conclusions. Simplification of the proof task is accomplished by subgoaling, which is the splitting of the theorem into a set of theorems,

eg. $\forall p.\ (h_1 \,\&\, h_2 \,\&\, h_3 ...) \rightarrow c_p$.

An example of one of these theorems and its proof appears in Appendix D.

The theorems generated by the proof methodology in part 4b, which relates the low-level and abstract-level exit assertions, also have the form that the conjunction of a number of low-level assertions implies the conjunction of a number of abstract-level assertions. Therefore, these proofs also use the subgoaling technique to arrive at theorems that are easier to handle.

Most of the theorems that result from applying the subgoaling technique are of the form:

Premise: $(a \rightarrow b)$
Conclusion: $(C \rightarrow D)$

where the premise is an implication dealing with low-level variables and the conclusion is an implication dealing with abstract variables. The proofs are generally obtained as follows. First, assumptions are made about the values of the abstract variables (i.e., assume C). The mapping function (rep_a) is then used to draw some conclusions about the values of the low-level variables. These conclusions are usually strong enough to imply the hypothesis of the low-level level implication (i.e., a). Next, since the premise states that a implies b and we know that a is true, the conclusion of the low-level implication (b) holds. Finally, the information about the value of the low-level variables that is stated in b is used to construct the abstract level values (using rep_a) that are needed for the conclusion of the abstract-level implication (D).

Essentially all of the mapping function work has been completed. Only a small part of the invariant proof (part 1) has been completed (about ten percent), and thirty to forty percent of part 4b, the exit specification implication, has been completed. These proofs are quite tedious but completely straightforward; automated tools would be a substantial help.

[13] The AFFIRM system is the successor to the XIVUS system referred to in §2.3.

3.3 Code-Level Verification

Parts 2 and 3 of the proof methodology described at the beginning of this section involve conventional inductive verification of procedural code with non-procedural specifications. Part 2, which deals with kernel initialization, is essentially just a special case of each of the proofs of the kernel calls (as described in part 3) and therefore is not discussed separately here. Little verification of part 2 was actually carried out.

Currently, verifying that code meets its specifications is costly, both in labor and computing power, even when state of the art automated verification tools are employed. Our experience in verifying portions of the kernel suggests where the major costs lie, and how they can be minimized. Those areas which significantly affected the task were program structure, verification system flexibility and user proof strategies.

The XIVUS semi-automated verification system [4, 34] used in the UCLA project was an ongoing research effort while the specifications were being developed. Using the XIVUS system, some verification of the consistency of those specifications with the kernel implementation was done, and many comments concerning areas where code-level verification costs could be reduced reflect those experiences.

Most of the overhead in the code-level verification process resulted from the cost of the theorem prover and the unfortunate necessity to repeat proof steps. Improvements in the theorem prover's power and user interface would of course be valuable. However, mechanisms are also needed to minimize unnecessary theorem prover invocations, which result either from redundant proofs or reverification.

Redundant proofs occur largely because common proofs are not identified and proven only once. As noted in §2.4, the verification of a given routine consists of several proofs, or verification conditions (VCs), one for each path through the code. Our experience indicates that often some of the VCs for a given routine are quite similar to one another. Sometimes they differ only in detail not relevant to their proof. For example, two VCs could be generated because of an if then construct that was not relevant to the assertions. The two VCs would have to be proven but one proof would be redundant. Mechanisms are needed to identify these situations, reduce the two VCs to a common form, and allow the proof to be done a single time.

A more subtle and yet much more common form of redundant proof can be seen by looking inside the various VCs for a given substructure. Often two entire VCs are not similar enough to allow a single proof but many of their subgoals (described in §3.2) are very similar and considerable savings could be realized if automated aids could take advantage of this.

Reverification is also inevitable as changes in code or specifications are made. What is needed is a means to minimize the resultant cost. Economies can be realized in several ways. For example, a system could be developed that limited the reverification of calling routines when the entry/exit specifications of a called routine change. Also, reverification of a modified routine would be cheaper if some of the original proofs could be shown to be still valid [3]. Savings within the reverification of individual routines are particularly significant, since small typographical errors in the specifications may be discovered only after several hours of computer time have been spent on the proofs. In general, then, a combination of administrative and technical tools is needed to localize the needs for reverification.

Less than twenty percent of the code-level proofs were completed. Because of the evolving state of the verification system while this work was in progress, accomplishing even this much was quite painful. Further, the remaining proofs require only additional tedious but straightforward effort; waiting for more effective machine aids seemed advisable once feasibility had been demonstrated. Assuming reasonable improvements, we estimate a few more months of effort would be required to complete the task.

4. Conclusions

The experience of addressing all aspects of specification and verification in a significant software project has led us to a number of conclusions.

First, it is possible to verify a realistic software system which had been designed to accomplish reasonably complex tasks reliably and with adequate performance; however, highly skilled researchers were needed in the system development, specification, and verification. A master's and Ph.D. thesis were produced [28, 11], and four to five man years were spent in the verification project reported here and in [22]. The effort involved in specification and verification overwhelms that involved in coding, although it is less than the total amount spent in design, implementation, and debugging.

Fortunately, much improvement is possible in both the skill required and the labor intensiveness of the project. Reengineering of the verification system would make the proof task considerably easier. Enhanced programming language scoping controls like import lists would help too.

Specification methods should exhibit considerable flexibility. For example, we found that our intrinsically interrelated data structures are not compatible with the concept of abstract data types. Organizing the low-level specifications by data structure is far preferable for people, and permits mechanical transformation into the more conventional call-by-call organization needed by verification systems. More development in specification frameworks is clearly desirable.

The recommended approach to program verification — developing the proof before or during software design and development — is often not practical. It is of course necessary to develop the system with verification goals clearly and continually in mind. The design and implementation proceeds, and it would be very costly to redevelop formal specifications and code verification each time. Further, there would be little gained, since nearly all the impact on software structure that would result from performing all the verification steps can easily be obtained without doing so. One can independently follow development practices which provide suitable structuring and clarity of code.

Verification of security properties differs from other verification problems. First, top-level specifications of the security properties are easier to develop because they are far simpler than a complete specification. We can ignore issues of scheduling and resource allocation, for example. If one were to compare our security criterion with the complete specification of the proper behavior of even a simple text editor, the ease of security specification would be even more apparent.

However, one should not conclude, as we initially expected, that it is possible to prove just those properties about the kernel needed to meet the security criterion. The policy manager depends on many properties of its process environment for its operation. If the kernel were to swap in a page of the policy manager, reversing the bits on the way, or incorrectly load the process' accumulators upon process invocation, then proofs of policy manager security would not be valid. This observation is not limited to our design as any realistic secure system contains trusted processes upon whose correct operation security depends. Therefore, while system specification is easier than many verification problems, it is not as simple as one might have expected.

Performance of the completed system is poor, an order of magnitude slower than standard Unix in some cases. However, the causes are either not related to the verification goal, or could be easily remedied through modest changes in the hardware base. There were two principal problems, language difficulties and hardware/software mismatches. As an example of the programming language problem, our Pascal compiler does not automatically generate machine code for array subscript bound checks, so that many inefficient source-code checks were instead coded in as procedure calls. Also, in-line procedure and function calls are not supported, which causes the majority of system execution time to be spent in procedure linkage overhead. Second, and more serious, is the domain switch overhead. A kernel-based architecture achieves its minimization of security-related code at the cost of frequent domain changes. In Data Secure Unix, domains are provided by the process mechanism. We were unable to reduce the process switch cost to a tolerable level, in part because of inadequate hardware support. Both of these problems can be substantially reduced by reasonable engineering of the programming language, its supporting compiler, and the hardware base.

In sum, there appears to be no technical reason, other than the necessity for engineering a suitable verification system, and exercising care in the hardware/software architecture match, that program proving methods could not be employed for the development of software where correct operation is critical. Secure operating systems appear to be an excellent case in point. Current techniques, however, are still not suitable for general use.

Appendix A: A Sample Code-Level Invariant

The specification below involves several different data structures, providing an example of the highly interrelated nature of the kernel data structures. The data structures are:

```
reloc_regs: A structure containing the hardware relocation register
        pairs for user and supervisor mode.
    var reloc_regs array [0 .. 31] of reloc_reg_entry;
    type reloc_reg_entry = record
        pdr: integer; Descriptive register with access and length
            fields. If the relocation register pair is not loaded,
            this register has a value of 0.
        par: integer; Address register.
    end.
```

```
reloc_index: A structure of indices relating the relocation registers
        to other structures.
    var reloc_index array [0 .. 31] of reloc_entry;
    type reloc_entry = record
        pgtbl: integer; If the relocation register is not loaded,
            this field has a value of -1. If the register is load-
            ed, this index indicates which page-frame is being
            accessed. Note that this field is redundant with
            par.
        cap: integer; If the relocation register is loaded, this in-
            dex indicates which capability the running process
            presented to gain access to the indicated page.
    end.
```

```
page: A structure with one entry per page-frame.
    var page array [0 .. num_pages -1] of page_entry;
    type page_entry = record
        id: object_id; Object id of the page in this slot.
        swapping: boolean; A boolean indicating if the page has
            completed swapping into memory.
        lock: integer; A counter of the number of ongoing
            I/O's involving this page-frame.
    end.
```

```
rpi: The index into the process table for the currently running
        process.
```

```
abstr_clist: A doubly dimensioned array of capabilities, one di-
        mension for each process and one dimension for the
        capabilities.
    var abstr_clist array [0 .. num_procs -1] of clist_entry;
    type clist_entry array [0 .. num_cap_entries -1] of record
        id: object_id; Id of the object.
        acc: access_type; Type of access allowed to the object.
        guess: integer; Non-security relevant efficiency field.
        info: integer; Field used by the policy manager.
    end.
```

```
proc_tbl_num_cblks: An array indicating the number of 512 byte
        blocks in the clist of each process. There are only two
        values, one for clists on small pages and one for clists
        on large pages.
    var proc_tbl_num_cblks array [0 .. num_procs -1] of integer;
```

Using *inbounds* and *subset* as predicates and *build_pdr* and *mem_blk_addr* as function abstractions, the parameterized invariant is:

```
iarr1(reloc_index, reloc_regs, page, abstr_clist, rpi, proc_tbl_num_cblks)=
∀ reg. ((   inbounds(reg, 32)
        ∧ (reloc_index[reg].pgtbl ≠ -1))
 → (inbounds(reloc_index[reg].pgtbl, num_pages)
 ∧ inbounds(reloc_index[reg].cap, proc_tbl_num_cblks[rpi]*cl_per_smpg)
 ∧ (reloc_regs.par[reg] = mem_blk_addr(reloc_index[reg].pgtbl))
 ∧ (not page[reloc_index[reg].pgtbl].swapping)
 ∧ (abstr_clist[rpi][reloc_index[reg].cap].id=page[reloc_index[reg].pgtbl].id)
 ∧ (∃ acc((reloc_regs.pdr[reg]=build_pdr(acc, reloc_index[reg].pgtbl, page))
 ∧ (subset(acc, abstr_clist[rpi][reloc_index[reg].cap].acc)))))
```

Informally, for all relocation registers in use, the following conditions must hold:

1. The page-frame index must be valid.
2. The capability index must be valid.
3. The address registers of the relocation register pair must have a value equal to the address of the start of the indicated page-frame.
4. The page in the page-frame must not be in the process of swapping into memory.
5. The id of the page in the slot must correspond to the id of the appropriate capability slot of the running process.
6. The descriptive register of the relocation register pair must:
   a. indicate a page length consistent with the page-frame size;
   b. indicate an access which is a subset of the access allowed.

Appendix B: A Sample Conditional Assertion

The conditional given in this appendix is part of a more general specification that ensures that pages in incore page-frames are reflected back to disk before the page-frame is reused. If the page has been modified. As an efficiency decision, pages not modified (referred to as clean pages) are not reflected. Pages can be modified either through I/O or via hardware instructions that execute through the relocation registers (defined in Appendix A). One of the bits in the *pdr* relocation register (namely the bit in position *pg_mod_bit*) indicates if the page has been modified while the relocation register has been pointing at it. The information in this bit must be saved before either element of the relocation register pair pointing at the page are modified since at this point the clean/dirty bit is automatically cleared. The information in this bit, together with I/O information about the page-frame, is stored in the *page_clean* array, which is declared below:

```
page_clean: An array of booleans storing information concerning
            whether the associated page-frame is clean (i.e. has the
            same value as is in its secondary storage copy)
    var page_clean array [0 .. num_pages -1] of boolean;
```

The actual conditional expresses what condition must exist if the relocation register has been modified during any given kernel call. Namely.

If, during any kernel call, the relocation register pair is modified, then,
  a) the relocation register was not previously in use;
or b) the page-frame has been or was already marked as dirty;
or c) the clean/dirty bit in the associated descriptor register indicated the page-frame was clean before the kernel call.

More formally:

```
crr2(reloc_regs, reloc_regs', reloc_index', page_clean) =
∀ reg. ((    inbounds(reg, 32)
        ∧ (reloc_regs.pdr[reg] ≠ reloc_regs'.pdr[reg]))
 → (    (reloc_index'[reg].pgtbl = −1)
      ∨ (not page_clean[reloc_index'[reg].pgtbl])
      ∨ (wrd_and(reloc_regs.pdr'[reg], pg_mod_bit) = 0)))
```

Appendix C: A Sample Mapping Function from Low-Level to Abstract

The *Incore-page-table* presented in §2.2 is constructed from four different arrays at the low level. The arrays are *reloc_regs*, *page*, *page_clean* and *reloc_index* and they are described in Appendices A and B.

The mapping function used to construct the *Incore-page-table* from the four arrays is *Rep-ict* which is defined as follows.

```
function Rep-ict(arr:array[0 .. 31] of reloc_reg_entry,
                 pg:array[0 .. num_pages −1] of page_entry,
                 pgcl:array[0 .. num_pages −1] of boolean,
                 rsi:array[0 .. 31] of reloc_entry):
                         array[0 .. Num-pages −1] of
                             record
                               Pgtbl: Id;
                               I/o-cnt: integer;
                               Clean: boolean;
                               Swapping: boolean
                             end;
entry: true;
exit: (∀i, 0≤i<Num-pages)
   (Rep-ict[i].Pgtbl = Rep-id(pg[i].id) ∧
   (Rep-ict[i].I/o-cnt = pg[i].lock) ∧
   (Rep-ict[i].Swapping = pg[i].swapping) ∧
   (Rep-ict[i].Clean = (pgcl[i] ∧ (∀j, 0≤j<32)
       ((rsi[j].pgtbl≠i) ∨ (wrd_and(arr.pdr[j].pg_mod_bit)=0))))
```

Thus,
Incore-page-table=Rep-ict(reloc_regs,page,page_clean,reloc_index).

Appendix D: A Sample Proof Relating Low-Level and Abstract Level Assertions

The following is an example of one of the theorems that is proved to satisfy step 1 of the proof methodology.

At the abstract level there is an invariant specifying that every non-free entry in the *Incore-page-table* is unique. (Note that in the proof below, the name *Incore-page-table* is abbreviated to *Ict*.) The low-level invariant that corresponds to this abstract invariant states that all non-null entries in the *page* array have unique id fields. The following theorem is used to verify that if this low-level invariant holds then the abstract invariant holds. The definition of the abbreviations, mapping functions and lemmas used in this proof are given in detail in [11].

Thm:

Premise.

```
(∀r, inbounds(r, num_pages))(∀rr, inbounds(rr, num_pages))
((obj_type(page[r].id)≠null)
      → (rr≠r  → page[rr].id≠page[r].id))
```

Conclusion:

```
(∀r1, 0≤r1<Num-pages)(∀r2, 0≤r2<Num-pages)
((In-use(r1) ∧ Ict[r1].Pgtbl=Ict[r2].Pgtbl)  → r1=r2)
```

proof:

1. $0 \leq r1 < Num\text{-}pages \wedge$ $0 \leq r2 < Num\text{-}pages \wedge$ $In\text{-}use(r1) \wedge$ $Ict[r1].Pgtbl = Ict[r2].Pgtbl$	1. Assumption
2. $Ict[r1].Pgtbl \neq Nullobj\text{-}id \wedge$ $Ict[r1].Pgtbl = Ict[r2].Pgtbl$	2. Abbr 10, 1 Subst, Simp
3. $Rep\text{-}id(page[r1].id) \neq Nullobj\text{-}id \wedge$ $Rep\text{-}id(page[r1].id) = Rep\text{-}id(page[r2].id)$	3. 1 Constr Ict, 2 Subst
4. $obj\_type(page[r1].id) \neq null \wedge$ $Rep\text{-}id(page[r1].id) = Rep\text{-}id(page[r2].id)$	4. 3 def *Rep-id*, Logic, Simp
5. $obj\_type(page[r1].id) \neq null \wedge$ $page[r1].id = page[r2].id$	5. 4 def *Rep-id*, Logic
6. $inbounds(r1,\ Num\text{-}pages) \wedge$ $inbounds(r2,\ Num\text{-}pages)$	6. Def inbounds, 1 Subst, Simp
7. $inbounds(r1,\ num\_pages) \wedge$ $inbounds(r2,\ num\_pages) \wedge$ $obj\_type(page[r1].id) \neq null \wedge$ $page[r1].id = page[r2].id$	7. Lemma 5, 6 Subst, 5 Adj
8. $r1 = r2$	8. 7 Premise, MT
9. $(\forall r1,\ 0 \leq r1 < Num\text{-}pages)$ $(\forall r2,\ 0 \leq r2 < Num\text{-}pages)$ $(In\text{-}use(r1) \wedge Ict[r1].Pgtbl = Ict[r2].Pgtbl \rightarrow r1 = r2)$	9. 1,8 UG

QED

In step 1, the hypothesis of the conclusion is assumed. In step 2 Abbr 10 which defines *In-use* is substituted into step 1 and the result is simplified. Step 3 uses the definition of the construction of the *Incore-page-table* and the fact that $r1$ and $r2$ are in the range 0 to *Num-pages* −1 to replace the values of *Ict* in step 2 with the mapping of the low-level values used to construct them. The conclusion of step 4 is derived from the fact that the constructed value is not the *Nullobj-id* and the definition of the *Rep-id* function. Step 5 uses the fact that since $obj\_type(page[r1].id) \neq null$ the part of the *Rep-id* function used behaves like a 1-1 function. Therefore, *Rep-id* can be removed from both sides of the equality. From the definition of *inbounds* and knowing the range limitations specified in step 1, the conclusion of step 6 can be drawn. Step 7 uses Lemma 5 which states that *Num-pages=num_pages* and the substitution rule to get the first two conjuncts of its conclusion. It then uses the adjunction rule and the result of step 5 to get its result. Using the results of step 7 and the premise, one can conclude that

$$r1 \neq r2 \rightarrow page[r1].id \neq page[r2].id$$

However, from the last conjunct of step 7 one knows that the conclusion of this implication is false. Therefore, by the modus tollens rule $r1=r2$. Finally, since in step 1 the ranges for $r1$ and $r2$ were assumed and no other specifications were placed on the range of these variables, by universal generalization the desired conclusion is derived.

Acknowledgments. The authors wish to express their appreciation to members of the UCLA Security Project, notably Charles Kline, Evelyn Walton, Mark Kampe, Allen Stoughton and Mike Urban. Without their contributions to the design and construction of the UCLA Data Secure Unix system, this report would not have been possible. In addition we wish to thank the reviewers of an earlier draft, particularly Butler Lampson.

Received August 1979; accepted October 1979; revised November 1979

References

1. Eggert, P.R., Hall, M., and Kemmerer, R.A. KAL KAN: An assertion language - - for the verification of korrect ALPO notation. Comptr. Sci. Dept. Memo 156, UCLA, Los Angeles, Calif., March 1976.
2. Gerhart, S.L., and Wile, D.F. Preliminary report of the delta experiment: Specification and verification of a multiple-user file updating module. Proc. Specification of Reliable Software Conf., April 1979, 198-211.
3. Gerhart, S.L. private communication.
4. Good, D.I., London, R.L., and Bledsoe, W.W. An interactive program verification system. *IEEE Trans. Software Eng. 1*, 1 March 1975, 59-67.
5. Hoare, C.A.R. An axiomatic basis for computer programming. *Comm. ACM 12*, 10 (October 1969), 576-583.
6. Hoare, C.A.R. Procedures and parameters: an axiomatic approach. *Lecture Notes in Mathematics 188*, E. Engler, Ed., Springer-Verlag, 1971, 102-116.
7. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica 1* (1972), 271-282.
8. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2* (1973), 335-355.
9. Jensen, K. and Wirth, N. PASCAL user manual and report. *Lecture Notes in Computer Science 18*, Springer-Verlag, 1974.
10. Kalish, D. and Montague, R. *Logic techniques of formal reasoning*. Harcourt, Brace, and World, Inc., New York, N.Y., 1964.
11. Kemmerer, R.A. Verification of the UCLA security kernel: Abstract model, mapping, theorem generation and proof. Ph.D. Th., UCLA, Los Angeles, Calif., 1979.
12. Lampson, B.W. Protection. Proc. 5th Annual Princeton Conf. on Information Science and Systems, Princeton, March 1971, 437-443.
13. Lampson, B.W. A note on the confinement problem. *Comm. ACM 16*, 10 (October 1973), 613-615.
14. Lampson, B.W., et al. Report on the programming language EUCLID. SIGPLAN Notices, 12, 2, February 1977.
15. Liskov, B., and Zilles, S. Programming with abstract data types. Proc. ACM Conf. on Very High Level Languages, SIGPLAN Notices, 9, April 1974, 50-59.
16. Liskov, B., and Zilles, S. Specification techniques for data abstractions. *IEEE Trans. Software Eng. 1*, 1 March 1975, 7-19.
17. Luckham, D., and Suzuki, N. Automatic program verification V: Verification-oriented proof rules for arrays, records and Pointers. Stanford Artificial Intelligence Laboratory Memo AIM-278, March 1976.
18. Musser, D. AFFIRM user's guide. USC/Information Sciences Institute, Marina del Rey, Calif. (forthcoming)
19. Parnas, D.L. A technique for software module specification with examples. *Comm. ACM 15*, 5 (May 1972), 330-336.
20. Popek, G.J. Protection Structures. *Computer 7*, 6 (June 1974), 22-33.
21. Popek, G.J., Kampe, M., Kline, C.S., and Walton, E.J. The UCLA data secure operating system. UCLA Technical Report, July 1977.
22. Popek, G.J. and Farber, D. A Model for verification of data security in operating systems. *Comm. ACM 21*, 9 (September 1978), 737-749.
23. Popek, G.J., Kampe, M., Kline, C.S., and Walton, E.J. UCLA data secure Unix. *Proc. 1979 NCC*, AFIPS Press, 355-364.
24. Ritchie, D.M., and Thompson, K. The Unix time-sharing system. *Bell System Technical Journal 57*, 6 (July-August 1978) 1905-1930.
25. Robinson, L., et al. On attaining reliable software for a secure operating systems. Proc. 1975 Int. Conf. on Reliable Software, April 1975, 267-284.
26. Robinson, L., and Levitt, K.N. Proof techniques for hierarchically structured programs. *Comm. ACM 20*, 4 (April 1977), 271-283.
27. Shaw, M., Wulf, W.A., and London, R.L. Abstraction and verification in alphard: Defining and specifying iteration and generators. *Comm. ACM 20*, 8 (August 1977), 553-563.
28. Walker, B.J. Verification of the UCLA security kernel: data defined specifications. Master's Th., UCLA, Los Angeles, Calif., October 1977.
29. Walker, B.J., Popek, G.J., and Kemmerer, R.A. Formalization of the top-level specification/verification of the UCLA security kernel. UCLA Computer Science Dept. Technical Report (forthcoming).
30. Walton, E.J. The UCLA PASCAL translation system. UCLA Computer Science Dept. Technical Report, January 1976.
31. Wells, R.E. Specification and implementation of a verifiable communication system. Master's Th., U. of Texas at Austin, December 1976.
32. Wirth, N. The programming language PASCAL. *Acta Informatica 1* (1971), 33-63.
33. Wulf, W.A., London, R.L., and Shaw, M. An introduction to the construction and verification of ALPHARD programs. *IEEE Trans. Software Eng. 2*, 4 December 1976, 253-265.
34. Yonke, M.D. The XIVUS environment: XIVUS working paper no. 1. USC/Information Sciences Institute, Marina del Rey, Calif., April 1976.

DATA BASE MANAGEMENT SYSTEMS SECURITY AND INGRES*

Deborah Downs+ and Gerald J. Popek

University of California at Los Angeles

Abstract

The problem of providing a secure implementation of a data base management system is examined, and a kernel based architectural approach is developed. The design is then successfully applied to the existing data management system Ingres. It is concluded that very highly secure data base management systems are feasible.

Introduction

The need for reliably implemented protection of access to data stored in integrated databases has been driven by a number of considerations:

a) Data handling accounts for the vast majority of computer activity, an estimated 85% [SenM69], with the continued existence of some organizations depending on their data management systems;

b) Some of the stored data is sensitive, and unauthorized access or modification would have serious consequences. The amount of such data will increase as computing is introduced to offices and organizations increasingly integrate their data;

c) Experiences in related areas, such as operating systems, suggest that waiting to solve the problem until the need is stronger may make cost effective solutions unavailable.

A very promising approach to secure systems design employs kernel architectures, which limit the amount of software upon which secure operation depends to a very small part of the overall system. Kernel architectures have been successfully applied to operating systems, but there are questions whether data management systems can be approached in a similar fashion. It has not been clear whether a kernel design for secure data management can developed which simultaneously dramatically minimizes the amount of code involved in protection, but still permits all necessary functionality, including support of data independence, flexible reorganization, good performance, and sophisticated query languages.

In this paper we will present a general design for a secure DBMS, together with a case study implementation. The design supports data security through the use of a kernel architecture which minimizes and encapsulates the software upon which correct protection enforcement depends. The design approach was applied to Ingres, an existing relational DBMS developed at U.C. Berkeley [HelG75]. An implementation, successfully carried out at UCLA, produced a modified version of the system, which is certifiably secure. The results of this effort lead us to conclude that secure data management is quite feasible.

2.0 Data Security and Kernel Architectures

To understand this project, it will be useful first to outline exactly what level of security is being supported. The solution presented here is not concerned with physical security but rather is a technological solution to security internal to the computer environment. The kernel design is used to support a data secure DBMS, as defined by Popek [PopG78b].

Briefly, the database is thought of as a collection of distinct, named *objects*. The DBMS supplies a collection of *operations* which users may apply to the objects. ***Protection data***, recorded in the system, states which users are permitted to access each object, and which operations they may employ. Formally stated assertions define data security more precisely.

It should be noted that the protection decision in this definition depends on the name of the data, not its value; therefore value dependent security decisions are not supported. Protected objects must also already exist in the data base as named entities, so statistical security will not be

*This research has been supported by the Advanced Research Projects Agency of the Department of Defense under Contract DSS MDA903-77-C-0211

+ Author's current address: University of Arizona, Department of Information Management Science.

supported. These more sophisticated definitions of security make the task of developing an implementation that reliably enforces protection controls considerably more difficult. Indeed it is not even clear how feasible the goal of a certifiable implementation would be. Fortunately, the more straightforward goal of reliable data security enforcement appears to provide much of the functionality seriously needed for secure applications. The issues in reliably supporting value dependent and statistical security are discussed elsewhere [DowD79].

The design for a secure DBMS presented here is based on a kernel architecture. Security kernels can be thought of as a structuring mechanism for any system concerned with security. All functions pertinent to security are included in the kernel while all functions not pertinent are excluded. Kernels were first used in operating systems and are reported in [WulW75], [PopG74], [PopG78a], [JanP76], [MilJ75], [SchM75]. This type of architecture is employed in the UCLA secure Unix design.

The main impetus for using a kernel based architecture is the difficulty of certifying the correctness of any large piece of code. Program verification, the method of certification with highest assurance, has only been completed successfully on a few thousand lines of code [RagL73], [GerS79]. Therefore it is very important to minimize the amount of code that must be certified. Another principle, "least common mechanism", also supports the use of a kernel architecture [Pop74b]. A common mechanism operates on behalf of many users, and may result in a security flaw if it operates incorrectly. In current DBMSs which try to offer some type of security, most functions of the DBMS are common mechanisms. Since little attempt was to be made to segregate the security mechanism in these DBMSs, correct operation of that security mechanism depended on the correct operation of the entire DBMS. In a kernel based secure DBMS the security kernel becomes the least common mechanism and is certified to support data security and thus prevent illegal modification or reference.

Support from a secure operating system is by necessity a prerequisite for a secure DBMS unless the DBMS takes over many of the functions of an operating system. The operating system supported environment required by our design of a secure DBMS includes:

1. a process environment that operates correctly and includes no trusted operating system code running in the process space,
2. correct identification of the user, and
3. data security enforcement among processes and data stored in files, including control over interprocess communciation, etc.

Items two and three above presumably are clear, but the first may bear some explanation. Data secure operating systems typically contain a small system nucleus (the kernel) responsible for security enforcement. The remainder of the operating system software runs encapsulated within each user process. No assurance regarding the correct operation of the encapsulated code is necessary to assure secure operation of the operating system, nor is it typically provided. As a result, a data secure operating system alone is not sufficient to support processes wishing to reliably enforce their own level of security. Not only must the operating system reliably protect the DBMS's data, it must also return correctly, exactly the data requested by a DBMS I/O call. Otherwise the DBMS cannot correctly enforce its own level of security since it cannot be sure what it has read or written. Further, any portions of the operating system's functions that execute in the process's environment must be certified for correctness since such code, if run in the same environment as the DBMS kernel, would have the DBMS security kernel's privileges. Portions of the operating system that may need to operate in the process environment include process initialization and resource management functions.

The secure Unix operating system developed at UCLA [KamM77], [KemR78], [PopG78c], [WalB77], [WalE76] is being used as an example secure operating system for the developement of the secure DBMS. At this time the operating system supports data security as well as the correct retrieval of data. Work is still needed on certifying the portions of Unix that reside in the process environment.

3.0 Secure DBMS Design

The following design, which was developed at UCLA on a secure Unix testbed, supports various grains of data security protection down to the size of domains in relations. In this paper all references to logical objects will be in terms of the relational model. However, the design is not restricted to any particular data model and other models would be equally appropriate.

Let us first look at a high level view of the system. Aside from the user software there are four main modules in the secure system architecture: the Kernel Input Controller (KIC), the base kernel, the format software and the Data Management Module (DMM). The DMM need not be certified but must run isolated in its own execution environment with no ability to return information to the user of the data base. The DMM contains all the non-security related portions of the data base management system and presumably with only a few changes could run alone as a non-secure DBMS. Included in the DMM would be such functions as parsing and decomposing queries;

choosing search strategies; creating and searching dictionaries, catalogues, and indexes; performing necessary joins or projections; compressing, encoding, or reformatting the data; etc. The KIC and the base kernel are the two functional modules of the DBMS security kernel and therefore must be certified. The KIC is the secure interface between the user and the DMM and the base kernel is the secure interface between the DMM and the data base. Format software, essentially run for each user with his privileges, performs some user specific data base actions, such as reformatting of data prior to its actual delivery to the user. A high level description of the functions of such a DBMS security kernel includes:

1. The kernel must assure at all levels of the DBMS a secure association between the data and its identification. If this association is not maintained there is no way to reliably determine access rights to the data.
2. Since the protection data is stated at the logical level and the actual access is taking place at the physical level, some secure translation process is necessary in order to compare the physical object to be accessed to the logical object specified in the protection data and in the original user request.
3. The DBMS kernel must enforce the security policy on each access, limiting the data being returned to the user to those specified in the protection data.

3.1 Retrieval

In order to see how the kernel design performs these functions, a sketch of both retrieval and update are presented. Figure 1 shows a retrieval operation.

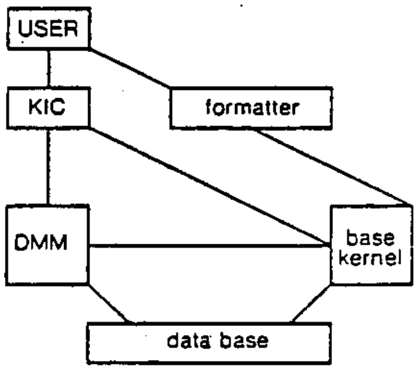


Figure 1

In this figure the four main elements of the design can be seen. Edges in the diagram indicate data flow. As already outlined, in order for the protection mechanism to be able to check a user's authorization for access, it is important that the identification of the data be correct. It is the KIC's function to supply this correct identification to the base kernel.

A user makes a request for retrieval of a certain piece of data and the KIC parses the request and retains the request type, in this case *retrieval*, and the logical names of each object to be retrieved. The following retrieval request in the language Quel used in Ingres [StoM76] will be an example query:

```
range of e is employee
retrieve (e.name, e.address)
where e.salary > 10000
```

The request asks that the names and addresses of employees who make more than $10,000 be retrieved from the employee relation. To begin execution the KIC would parse the above request enough to retain "retrieve, employee.name, employee.address". (In the relational model both the relation name and the domain make up the logical entity name of an object.) But the logical name specified in the query may not be the system wide logical name but merely a name used in one user's schema or "view". To reduce a multiplicity of logical entity names used to identify the same data with a single system wide logical name, the KIC must also parse the data definition language and build the tables necessary to translate a user schema name or view to a system wide logical entity name. The KIC would also parse the qualification clause, the "where" clause, and check the protection data to determine whether the user held legal read access to employee.salary, employee.name and employee.address. Illegal queries would be aborted before any further processing. Note that the KIC checks the qualification clause to restrict the user from retrieving illegal information by qualifying retrieval of legal data objects by the value of illegal data objects. If the request is legal, the KIC then passes to the base kernel the type of command, the system wide logical entity names, and the user identification supplied by the secure operating system. In the Ingres retrieve request above there is no user schema or view so the logical names present in the query are the system wide logical names (employee.name, employee.address).

Meanwhile the request is also being processed by the DMM. The DMM performs all the normal functions of a data base management system: it makes the translations from one level to the next, follows access paths, chooses the access methods, records performance and usage statistics, does any necessary locking, etc. Eventually the DMM needs to access the data base itself.

It is not really necessary that the DBMS kernel check every read access made to the data base by the DMM. Only data actually returned to the user must be checked. In the Ingres retrieval example above all tuples in the "employee" relation will be read, but only four will be returned to the user. In more complicated queries where "joins" are necessary, a very small percentage of the read accesses may return data to the user. Having the DBMS kernel involved in every read access could degrade performance considerably. Data security enforcement is not weakened if the DMM has unlimited read access to the data base so long as the DMM is properly encapsulated by the operating system. The secure operating system would enforce the read only access restriction for the DMM along with blocking the DMM from returning any data directly to the user.

When tuples are selected which answer the request and are to be returned to the user, the DMM prepares an operating system read command specifying the physical location that is to be accessed and that command is passed to the DBMS base kernel. All data to be returned to the user is retrieved by the DBMS kernel. If the DMM should not invoke the DBMS kernel when data should be returned to the user, the only result is that the user does not receive the data, which does not violate data security. In the Ingres retrieve example there would be one read command from the DMM for each tuple in the "employee" relation that matched the qualification.

Before retrieving information in the data base the DBMS base kernel must check to see if the user is allowed to access the data in the location specified by the DMM. The request from the DMM can include both the physical location and the logical entity name. Of course the correlation of these parameters cannot be trusted so far as security enforcement is concerned, since to do so would imply that the DMM is trustworthy. Recall that to support data independence the protection data maintained by the DBMS base kernel identifies the objects by their system wide logical entity name. Therefore some form of reliable mapping is still necessary between the DMM's physical location specification and the protection data's logical identification.

In this general design a physical to logical mapping is used which takes the form of a tag on each piece of separately protectable data in the data base [FriT70]. This tag is maintained completely by the base kernel and the DMM has no knowledge of it. The tag is an encoded form of the logical entity name of the data and its size depends on the number of distinct logical entities to be protected. The information received from the KIC: logical entity name, command type, and the user identification (which must be obtained from the secure operating system), is used by the base kernel to check if the specified user is allowed access to the logical entity. In the retrieve example the user's access to the "employee" relation and the "name" and "address" domains must be checked. When the read command is received from the DMM, the base kernel reads the data along with its tag from the data base in the location specified by the DMM and then, using that tag, checks to see if the logical entity is the same as was specified by the KIC. In our example request the logical entities are "employee.name" and "employee.address". If the tag retrieved from the data base matches the information supplied by the KIC, the base kernel returns the results directly to the user without allowing the DMM to manipulate the data.

The data to be returned to the user may have to be translated from the form used for storage in the data base to the user schema's format; also the requested domains must be separated from any other domains in the tuple. Choosing the correct domains is security relevant since the user may not have legal access to other domains in the tuple, and must be done by the kernel. However, the correctness of any necessary manipulation of the chosen domains, such as decoding or formatting, is not relevant for data security purposes and can be done by untrusted code, so long as no other users' data influence the operation of this function. Therefore, a formatter module is supplied, that runs in its own execution environment protected by the operating system, with a new copy used on each invocation so that data from prior runs is not retained. The formatter module is the site where all postprocessing of retrieved data would be done, including joins of retrieved tuples, for example. Note that the operation that parses the data definition language, where the user specifies user schema formats for relations and domains, and prepares the tables which the data formatter uses; must also be run under the same confinement constraints in order that those tables also be free from retained user data. After the data is formatted in terms of the user's schema, it is delivered to the user workspace. In the Ingres retrieval example the results might look like:

name	address
John Williams	7746 Hillburg, L.A., Ca
Hope Botts	R#2, Conte, Ohio
Elmer Crep	Hill Ave, N.Y. New York
Jamie Cann	3837 E 111th, Hoare, Virginia

3.2 Update

Now let us look at an update operation, referring again to Figure 1. An example request for update, again using the Ingres format, might be:

```
range of e is employee
replace e (salary = e.salary * 1.1)
where e.salary < 10000
```

This request asks that all employees in the employee relation who make less than $10,000 get a 10% raise. When the request comes from the user the KIC parses it, looking at this stage for any requests which would change the contents of the data base, such as creates, updates or deletes. Whenever the KIC recognizes such a request it copies the logical name, the data, and the command type, again checking read access to any qualification clause data objects and write access to the target data objects. Any illegal access will cause the request to be aborted. In the replace example the KIC would retain "replace, employee.salary" so that the logical name can be translated to the system wide logical name. This translation must be done correctly for security to be maintained. This example also illustrates the fact that for many query languages the KIC parse task is considerably simpler than the full scale parse which the DMM must perform.

The data often may have to be translated from the user schema format to the form maintained in the data base. Packing is just one example of possible data conversions used. This step is done by the untrusted but isolated data formatter in a manner similar to that described above in retrieval, but in reverse. Thus the KIC, after invoking the data formatter, directly passes to the base kernel the system wide logical entity name, the translated data, the command type, and finally the user identification as supplied by the operating system. In our example the data to be stored in the data base is a computation on the data already in the data base. Therefore, after the data is retrieved and passed to the formatter, it will perform the multiplication "employee.salary * 1.1".

Meanwhile the request has also been processed by the DMM analogously to the retrieval case. The DMM eventually is ready to execute a write on the data base and it prepares a request specifying the physical location. In our example many tuples may be updated. The KIC checked the user's update access to the logical entity name as specified in the protection data.

When the base kernel receives the request from the DMM selecting the location to be updated, it reads the specified location in the data base and checks the logical entity tag on the data to be sure that the DMM has pointed to the correct logical entity. If the location is correct and if the user is allowed update access to the logical entity, the data that was computed by the formatter is written with the correct tag attached. The base kernel must place the domains in the correct location in the tuple. On a create request the base kernel would have checked to see if space were really available, and then created the tag from the information received from the KIC. The DMM is never allowed to manipulate either the tag or the data that is stored in the data base.

This design supports value independent data security because access authorization is checked on any attempted access to the data base where data is returned to the user for retrieval or received from him for update. The identification of the data is always secure since the kernel maintains the association between the data and its identification during translation through all levels. Most important, the mechanism doing the checking of all relevant stages of the operation and maintaining the association can be certified.

4.0 Ingres

It is instructive to apply the design just outlined to a real DBMS. One might prefer to design a secure DBMS from scratch with the security kernel as the design base and security the guiding concept during the design and implementation. However, much effort has already been expended on existing DBMSs and it would certainly be helpful if the security kernel could be retrofitted to some already existing DBMSs. The Ingres system, a relational DBMS developed at U.C. Berkeley which runs on the Unix operating system, was examined to demonstrate the possible feasibility of such a project. Ingres was chosen for several reasons:

1. The computer facility most easily available for the sample implementation ran Unix, and Ingres was already installed.
2. Ingres is available free for Universities.
3. The relational model which Ingres supports contributed to an internal structure that eased the retrofit effort.
4. Ingres has been programmed in a structured manner. The internal modularity of the code was essential to the success of the alterations that were performed.
5. A secure version of Unix has been developed at UCLA, providing a suitable testbed.

There are also disadvantages in using Ingres. It does not have a wide use as an commercial system, although it has been installed in over 100 installations [EpsR78]; the relational model is not used in many commercial DBMSs; and there is hardly any documentation of the internals of Ingres other than the commented code itself.

Overall the advantages far outweighed the disadvantages and so Ingres was chosen.

4.1 Ingres Internals

Ingres is made up of 5 major modules: Monitor, Parser, OVQP, Decomp and DBU. These modules are organized into processes and Figure 2 shows a representation of the process structure, including communication channels, for release 6.2, which was used for the security alterations. The Monitor handles the terminal interactions with the user. The Parser parses the user query and contains the lexical analyzer, parser and concurrency control routines. OVQP, the one variable query processor, does accessing of tuples from a single relation given a particular one variable query. Decomp decomposes queries involving more than one variable into a series of one variable queries and accumulates the results until the query is satisfied. DBU, the data base utility, contains all the auxiliary processing such as creating databases, sorting, handling expiration dates, etc. Access Method software (AM), present in several modules, provides low level access to actual tuples in the database. Ingres is a significant DBMS, with over half a million lines of compiled code. Since Unix limits a process to 64K words of code, dividing Ingres into several processes was necessary. The DBU, which in itself is much larger than 64K, runs with many overlays.

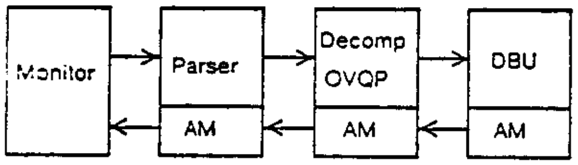


Figure 2

Ingres supports the language Quel for interactive access to the data base. In this paper, Quel has been used for examples, and explained as necessary. A full presentation of the language can be found in [HelG75].

4.2 Retrieval in Ingres

Operation of Ingres can be demonstrated by running through a simple retrieval request.

```
1   range of p is parts
2   range of s is supply
3   retrieve (p.pname, s.quan)
4   where p.color = "black" and p.pnum = s.pnum
```

This request requires a join on the parts and supply relations in order to retrieve the part name and the quantity of black parts. Statements 1 and 2 are the range statements and specify what relations must be accessed to retrieve the query results. In this case parts and supply are two relations already existing in an Ingres data base. Statement 3 specifies what domains are to be retrieved and which relations they are located in; the letters preceding the domain names specify the relations. This part of the query is called the target list. Statement 4 is the qualification list and specifies conditions to be met by a tuple before it is retrieved. In this case the color of the part must be black and a join of the two relations parts and supply is made on part number.

Ingres will begin execution of the query in the Monitor where a workspace will be created for the user. Interaction between the user and the Monitor continues until the query has been formulated correctly. When the syntax is correct the query is sent to the parser. There the query is parsed and the results sent to Decomp.

There are several important tables involved in servicing a query. These tables, as with all information in Ingres that is variable, are retained in the data base as relations. The first table is the "relation" relation. The "relation" relation contains one tuple for each relation in the data base. Stored in this tuple is the relation-name, information which describes how each relation is actually stored in the data base, ownership information, size data, number of domains, etc. The second important table is the "attribute" relation which contains one tuple for each domain of each relation in the data base. This relation describes the position of each domain in the tuple, its format, its length, whether it is used in part of the key, and whether or not it is coded. The "relation" and "attribute" relation are used by Decomp and OVQP to access the other relations in the data base.

To continue the execution of the retrieval command Decomp decomposes the query into one variable queries. "One variable query" refers to the variables in the range statement that label relations. In the example of Figure 2 the first variable "p" labels parts and the second variable "s" labels supply. Decomp must first formulate a one variable query that retrieves all the domains labeled by one variable (p) and then use those retrieved domains to formulate other one variable queries which retrieve the domains labeled by the other variable (s). The first one variable query that would be formed for the request in Figure 2 is shown in Figure 3. Ingres uses a tree that is an endorder representation of the request, with the target list on the left branch of the ROOT node and the qualification list the right branch. RESDOM's are result domains and in this tree indicate what results are to be returned to the user by the variable nodes which are the RESDOM's right child. The variable nodes contain pointers to the correct member of the "relation" relation and also to the correct member of the "attribute" relation along with format information. The end of the target list is designated by the TREE node. Qualification clauses, the right branch off the ROOT node, are headed by an AND node and the last qualification is marked by QLEND.

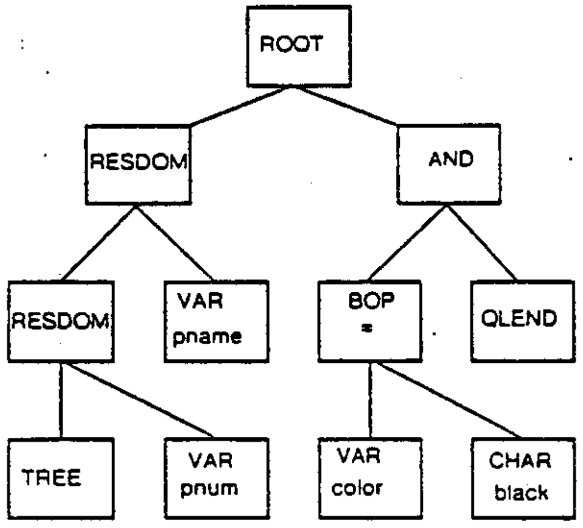


Figure 3

Decomp attempts to chose first the one variable qualification that would most reduce the number of tuples selected. In this case the qualification that "part color be black" was chosen and is represented in the qualification clause. This request tree, as with all one variable requests sent to OVQP, only specifies one relation, in this case parts, and the "one variable" is "p". The target list contains RESDOM nodes and VAR nodes for those domains to be retrieved from the part relation. In this case the part number (pnum) must be retrieved for the join with the part number in the supply relation. The part name (pname) is retrieved in order to eventually return it to the user. The complete algorithm used by Decomp for chosing one variable queries to be passed to OVQP is given in [StoM76].

Decomp must now send the one variable query shown in Figure 3 to OVQP. Decomp gives to OVQP a representation of the tree and specifies whether the results are to be placed in a specified temporary relation or returned to the user. For the one variable query in Figure 3 the results will be placed in a temporary relation.

OVQP will retrieve those tuples whose part color is black and will save their pnum and pname in a temporary relation. OVQP will inform Decomp when processing of the one variable query is finished and Decomp, who has retained the name of the temporary relation where the results are stored, can continue processing the query using the intermediate results in the temporary relation. Figure 4 shows the results that would be placed in the temporary relation if the query in Figure 3 were invoked.

pname	pnum
disk	3
tape drive	4
paper tape roader	13
paper tape punch	14

Figure 4

Decomp must then use the information in the temporary relation and the other qualification clause which references a domain in the supply relation to form other one variable queries to be sent to OVQP. Figure 5 represents the tree formed for the first one variable query on the supply relation.

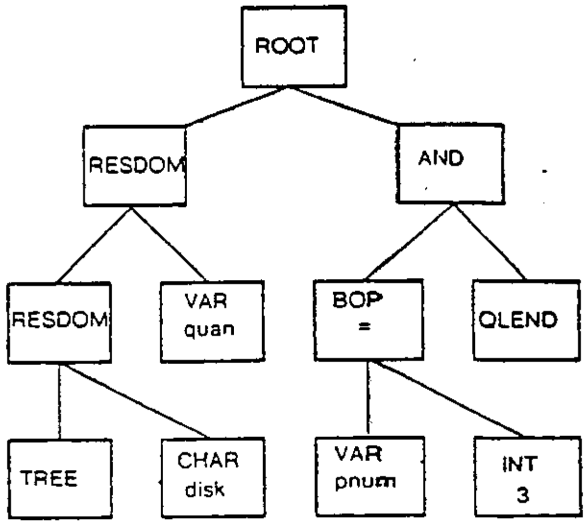


Figure 5

In this tree representation of the one variable query, Decomp replaces the domain names in the parts relation with the values retrieved in the previous one variable query and stored in the temporary relation. The query indicated by this tree will be repeated and sent to OVQP for each of the four tuples stored in the temporary relation shown in Figure 4, with the new values substituted for the part relation's domains. The target list is the same target list as the original query, since the part *pname* and the supply *quan* are to be returned to the user. But since pname came from the parts relation and was retrieved and stored in the temporary relation, the actual value for pname, "disk", has been inserted in the target list. The value for the other result domain must still be retrieved from the supply relation. The qualification for finding the correct tuple in the supply relation is that the pnum be 3, where the value 3 was also obtained from the temporary relation.

Since the final values will be retrieved on this round, OVQP is instructed by Decomp to return the results specified by the target list to the user. After the tuple is retrieved from the supply relation with the part number (pnum) = 3, a tuple containing the part name and the quantity are returned to the user. All tuples in the supply relation are checked for pnum = 3. After the tuples are tested and any results returned to the user, Decomp is signaled and another tree is formed to be sent to OVQP. The results that are printed on the users terminal by OVQP after processing the four supply trees are shown in Figure 6.

pname	quan
disk	2
disk	3
disk	1
tape drive	1
tape drive	6

Figure 6

One line was printed for every tree sent to OVQP that specified sending the results to the user. In the case of this retrieval request there were several instances of disks and tape drives that were black in the supply relation, but there was no supply of black tape punches or paper tape readers.

Updating in Ingres is implemented in a manner similar to retrieval except that the final results are written into the data base instead of being returned to the user.

5.0 Ingres Security Kernel

Before outlining the security kernel structure necessary for Ingres it is important to list what functions a security kernel must supply to support retrieval and update. For retrieval, it is necessary to:

1. Parse the query.
2. Check access to relation.domains in the qualification clause (it may be efficient to check access to relation.domains in the target list also) and cancel the request if access is denied.
3. Upon an Ingres request to return data to the user, check access to the relation.domains requested; if the retrieval is legal, cause the I/O, retrieving the data and passing it to the formatter for forwarding to the user. If it is not legal, cancel the request.

In the general design presented previously the first two functions are handled by the KIC and the last one by the base kernel. In the Ingres implementation of the security kernel, the KIC and the base kernel are an integrated module and will be referred to as the security kernel. Merging was done in order to minimize process switching.

For an update request the security kernel must:

1. Parse the query.
2. Check the legality of access to relation.domains in qualification clause (and perhaps target list) and cancel request if it is not legal.
3. Retain the relation.domain names with the data to be placed in the data base.
4. If the command is *replace* the tuple must be retrieved from the data base, the correct domains updated with the retained data and the tuple rewritten after the legality of the access has been checked. If the command is *append*, the tuple is formatted and written where Ingres specifies. If the command is *delete* the tuple is deleted.

The security kernel for Ingres will run in a separate process so that the secure operating system can protect the DBMS security kernel's environment. The process structure for Ingres release 6.2 with the security kernel is shown in Figure 7.

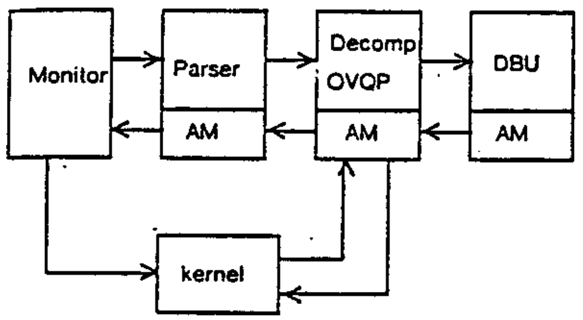


Figure 7

Let us follow a retrieval in detail to see how the security kernel operates. When the user starts Ingres an invocation of the DBMS is initiated for her/him; all users have their own family of Ingres processes. The user enters commands into a workspace, in actuality a file which is writeable only by the user and will be read by Ingres and the security kernel. The Monitor communicates with the user until the query in the workspace is in the correct form. Since the Monitor cannot access the data base and since a new copy is invoked for each user the Monitor cannot retain data. Therefore the Monitor actions are not security relevant and need not be verified. When the query is in the correct form the security kernel will receive a copy, and parse it. Using the query that was shown in Figure 2 as an example, the security kernel would request the user's id from the secure operating system and check the protection data to assure that the user held read access to part.color, part.pnum, part.pname, supply.pnum, and supply.quan. If read access is not allowed to all domains, the query is cancelled and the user is informed.

During this time Ingres has also parsed the query and Decomp has built the query tree. However, there is a problem in checking access to a domain at the time when the retrieved domain is to be returned to the user. In a query with more than one variable, intermediate data has been retrieved and stored in a temporary relation. When the data is to be returned to the user it has already been substituted into the tree and there is no indication of the tuple or relation where it originated. In the earlier example, after the parts relation has been retrieved and its values substituted into the tree, there is no way to determine from which tuple in the parts relation the value "disk" was retrieved. In the secure architecture however, it is necessary that the security kernel itself actually retrieve the tuple and select the proper domain to obtain the data, since Ingres cannot be trusted to have placed the correct value in the tree.

It would be helpful if Ingres instead of actually retrieving the tuple would instead retrieve a pointer to the tuple and retain that pointer in its temporary relation and in the tree. Ingres defines its relations in the "relation" relation, its domains in the "attribute" relation, and its tuples by a "tid" (tuple identification). The tuples in each relation are numbered sequentially beginning at zero. The tid is considered to be in domain 0 of every tuple in every relation. The tid can be the pointer to be saved in temporary relations and inserted in the tree by Ingres. Then using the tid and the pointers to the "relation" and "attribute" relations the security kernel can retrieve the tuple itself and pick the correct domain. Since the VAR node already contains a pointer to the "relation" relation it is only necessary that the security kernel change the domain pointer to 0 and also alter the relevant format information in order to force Ingres to retain the tid of the tuple instead of the actual domain. If Ingres should chose the wrong tid, that would not be a data security relevant failure, since protection is at the level of domains and not tuples. The domain the user receives may not answer the qualification clause but it will still be a domain to which the user has legal access. At the point where the security kernel is called, when the actual data is to be returned to the user, the tree contains tids—or pointers—to the tuples which hold the information that is to be returned to the user.

In order to force Ingres to retrieve tid's instead of values, the security kernel is called after the tree has been formed in Decomp. The security kernel makes a copy of the target list, which would be the left branch of the tree from the ROOT node as shown in Figure 3, to retain itself. The target list at this time specifies in the VAR nodes which relation, which domains, and the correct format and position in the tuple. The original tree's target list is then altered to request tids instead of values and sent on for processing by Ingres. At this point the target list could also be checked for correspondence to the original user's query if the domains in the target list were retained when the sercurity kernel parsed them. If a correspondence check is made, the security kernel can assure that the requested domains are retrieved. Otherwise the security kernel can only assure that legal domains are retrieved.

When OVQP receives a tree specifying that the results are to be sent to the user, the security kernel will be called for each RESDOM and VAR combination and will be passed the tid of the tuple containing the domain to be retrieved. Using the information in the retained copy of the original tree, which was checked by the KIC for security authorization according to the protection data, plus the tid, the tuple is retrieved and the domain formatted and printed out.

To print out Figure 6 the kernel would have to be invoked eight times, once for each domain in each tuple. But Ingres, in producing the output in Figure 6, accessed the data base a total of 106 times to read the tuples in the parts and supply relations. These numbers result when the query is run with the standard demonstration relations stored in the database supplied with an Ingres release. Parts had 14 tuples and each was accessed once by the query shown in the tree in Figure 3. The supply relation has 23 tuples and all 23 were accessed for each of the four trees represented by Figure 5. These accesses do not include any made by Ingres to check indexes or other secondary information. The extra accesses necessary for the security kernel should therefore be a small increase in the total. For update the security kernel functions are similar to retrieval.

5.1 Comments

The size and complexity of the security kernel needed to implement the mainline functions of retrieval and update as described here are surprisingly limited: about 750 lines of the high level langauge C. Since some of that code was not built with kernel architecture goals in mind (see below), one expects that further reductions in kernel size and complexity are possible.

In the actual implementation, rather than tagging, the low level access method software of Ingres was included in the kernel to assure the correctness of that portion of the logical/physical map. This method, while simpler in the case of Ingres than the general tagging method previously outlined, has the disadvantage that additional access methods or changes to the existing ones would requre kernel alterations, together with recertification efforts.

There are other functions of a DBMS which affect security that were not discussed and that are not supported by the kernel which has been implemented: back-up and recovery, a general security policy, etc. A general approach for these facilities that is an extension of the basic technique presented here is outlined by Downs [DowD79].

6.0 Conclusions

This paper has presented a general design for a data secure DBMS that can be used to develop a safe environment for handling sensitive data. We have also shown how this design could be retrofit to an already existing system, resulting in the first DBMS with a high degree of data security. We feel that the relatively easy retrofit of the security kernel architecture to Ingres illustrates feasibility both as a general security design and as a retrofit package in some cases. However other DBMSs may not prove to be as applicable to retrofit, since Ingres was definitely a very structured, modularized system.

The promise of kernel based architectures in data management is both surprising and encouraging. Arguments had been raised why it would not be feasible [StoM78], but without limiting the amount of mechanism involved in protection, it is not clear how reliable enforcement could be accomplished.

Acknowledgement

It is a pleasure to acknowledge Robert Epstein's help in the design of alterations to the actual Ingres implementation.

7.0 References

[DowD77] Downs, Deborah and Popek, Gerald J. "Approaches to Data Management Security," (abstract) Workshop on Operating Systems and Data Bases, Northwestern University, Illinois, March 1977.

[DowD79] Downs, Deborah. Data Base Management System Security. Ph.D. dissertion. Dept. of Computer Science, UCLA, 1979.

[EpsR78] Epstein, Robert. (personal communication), member U.C. Berkeley Ingres Project. December 1978.

[FriT70] Friedman, T.D. "The Authorization Problem in Shared Files." IBM Systems Journal, Vol. 9, No. 4, 1970.

[GerS79a] Gerhart, S.L., D.F. Wile. "Preliminary Report of the Delta Experiment: Specification and Verification of a Multiple-User File Updating Module." Specification of Reliable Software Conference. Cambridge Mass., 3-5 April 1979.

[HelG75] Held, G.D. et al. "INGRES- A Relational Data Base Management System." Proc. AFIPS 1975 NCC, Vol.44, AFIPS Press, Montvale, N.J., 1975, pp.409-416.

[JanP76] Janson, P.A. Removing the Dynamic Linker from the Security Kernel of a Computing Utility, MIT, Masters Thesis, MAC TR-132, June 1974.

[KamM77] Kampe, M. et al. "The UCLA Data Secure Unix Operating System Prototype," UCLA Computer Science Technical Report (unpublished), July 1977.

[KemR78] Kemmerer, R. Verification of the UCLA Security Kernel: Abstract Model, Mapping Functions, Theorem Generation and Proof, Ph.D. Thesis, Computer Sci. Dept., UCLA, 1978.

[LamB71] Lampson, B.W. "Protection," Proceedings Fifth Princeton Symposium on Information Sciences and Systems , Princeton University, March 1971, pp.437-443.

[MilJ76] Millen, J.K. "Security Kernel Validation in Practice," Communications of the ACM, Vol.10, No.5, May 1976, pp.243-250.

[PopG74] Popek, G.J. "Protection Structures," IEEE Computer, Vol.7, No.6, June 1974, pp.22-33.

[PopG74a] Popek, G. and Kline, C. "Verifiable Secure Operating Systems Software," Proc. AFIPS 1974 NCC, AFIPS Press, Montvale, N.J., pp.135-142.

[PopG74b] Popek, G. "Principle of Kernel Design," Proc. AFIPS 1974 NCC, AFIPS Press, Montvale, N.J., pp.977-978.

[PopG78a] Popek, Gerald J. and Kline, Charles S. "Issues in Kernel Design," Proc. AFIPS 1978 NCC, AFIPS Press, Montvale, N.J., pp.1079-1086.

[PopG78b] Popek, Gerald J. and Farber, David A. "A Model for Verification of Data Security in Operating Systems," Communications of the ACM, Vol.21, No.9, September 1978, pp.737-749.

[PopG78c] Popek, Gerald J. et al. "UCLA Secure Unix," Proc. AFIPS 1979 NCC, AFIPS Press, Montvale, N.J., pp.355-364.

[RagL73] Ragland, L.C. A Verified Program Verifier, Ph.D. Thesis, University of Texas, Austin, 1973.

[RitD78] Ritchie, D.M. et al. "Unix Time-Sharing System," The Bell System Technical Journal, Vol.57, No.6, Part 2, ISSN0005-8580, July-August 1978.

[SalJ74] Saltzer, J.H. "Protection and the Control of Information Sharing in Multics," Communications of the ACM, Vol. 17, No. 7, July 1974, pp.388-402.

[SchM75] Schroeder, M. "Engineering a Security Kernel for Multics," Proc. ACM 5th Symposium on Operating Systems Principles, ACM Operating Systems Review, Vol.9, No.5, November 1975, pp.25-32.

[SenM69] Senko, M.E. "Information Storage and Retrieval Systems," Advances in Information Science , Vol.2, Ed. J.T. Ton, Plenum Press, New York, 1969, p.230.

[StoM76] Stonebraker, Michael, et al. "The Design and Implementation of INGRES," ACM Transactions on Database Systems, Vol.1, No.3, September 1976, pp.189-222.

[StoM78] Stonebraker, Michael, private communication, 1978.

[WalB77] Walker, B. Verification of the UCLA Security Kernel: Data-defined Specifications, Masters Thesis, Computer Sci. Dept., UCLA, October 1977.

[WalE76] Walton, E. "The UCLA Pascal Translation System," UCLA Computer Science Dept. Technical Report (unpublished), January 1976.

[WesA68] Westin, A. Privacy and Freedom, Athenaum, N.Y., 1968.

[WulW75] Wulf, W., et.al. "HYDRA: The Kernel of a Multiprocessor Operating System," Communications of the ACM, Vol.17, No.6, June 1974, pp.337-345.

Encryption and Secure Computer Networks*

Gerald J. Popek and Charles S. Kline

University of California at Los Angeles

Abstract

The increasing growth of computer networks and general distributed computing applications, together with concern about privacy, security and integrity of information exchange, has created considerable interest in the use of encryption to protect information in computer networks. This interest has been accompanied by the development of various encryption algorithms.

Here issues of how to make use of encryption in networks are discussed. Key management, network encryption protocols, digital signatures, and a comparison of the utility of conventional and public key encryption are all considered, together with related issues.

Key words and phrases: computer networks, computer security, encryption, public key cryptosystems, digital signatures, network registries, encryption protocols

CR categories: 3.9, 4.35, 4.39, 5.39, 6.29

1. Introduction

It has long been observed that, as the cost per unit of equivalent computation in small machines became far less than large centralized ones, and as the technology of interconnecting machines matured, computing would take on a more and more distributed appearance. This change of course is now happening. In many cases, users' data manipulation needs can be served by a separate machine dedicated to the single user, connected to a network of integrated data bases. Organizational needs, such as easy incremental growth and decentralized control of computing resources and information, are also well served in this manner. Multiprogramming of general application software in such an environment diminishes in importance.

As a result, the nature of the protection and security problem is beginning to change. Concern over the convenience and reliability of central operating system protection facilities is transferring to analogous concerns in networks. The issues of protection in computer networks differ in several fundamental ways from those of centralized operating systems. One of the most important distinctions is the fact that the underlying hardware cannot in general be assumed secure. In particular, the communications lines that comprise the network are usually not under the physical control of the network user. Hence no assumptions can be made about the safety of the data being sent over the lines. Further, in current packet switched networks, the software in the switches themselves is typically quite complex and programmed in assembly language; one cannot say with certainty that messages are delivered only to their intended recipients.

The only general approach to sending and storing data over media which are not safe is to use some form of encryption. Suitable encryption algorithms are therefore a prerequisite to the development of secure networks, and considerable work has been developing in this area. However, equally important questions concern integration of encryption methods into the operating systems and applications software which are part of the network. We focus here on these latter issues, taking a

* This work was supported by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211.

pragmatic, engineering perspective toward the problems which must be settled in order to develop secure network functions. Cases where the safety of the entire network can be assumed are not discussed here, because issues in that environment are essentially those of distributed systems alone.

In networks, as in operating systems, there are several major classes of protection policies that one may wish to enforce. The most straightforward policy, satisfactory for most applications, concerns data security: assuring that no unauthorized modification or direct reference of data takes place. Highly reliable data security in networks today is feasible; suitable methods to attain this security will be outlined in the next sections.

A more demanding type of policy is the enforcement of confinement in the network: preventing unauthorized communication through subtle methods such as signalling via noticeable variations in performance [LAMP73]. One commonly mentioned (and fairly easily solved) confinement problem is traffic analysis: the ability of an observer to determine the various flow patterns of message movement. However, evidence to be presented later indicates that the conditions under which confinement in general can be provided in a network are quite limited.

In the following sections, we describe design problems and alternatives available for the design of secure networks, discuss their utility with respect to data security and confinement, and present an illustrative case study. The material is intended as a practicum for those concerned with the development of secure computer networks, or those who wish to understand the characteristics of encryption algorithms useful in network applications.

1.1. The Environment and its Threats

A network may be composed of a wide variety of nodes interconnected by transmission media. Some of the nodes may be large central computers; others may be personal computers or even simple terminals. The network may contain some computers dedicated to switching message traffic from one transmission line to another, or those functions may be integrated into general purpose machines which support user computing. One of the important functions of computer networks is to supply convenient, private communication channels to users, similar to those provided by common carriers. The underlying transmission media, of course, may be point-to-point or broadcast. Considerable software is typically present to implement the exchange of messages among nodes. The rules, or *protocols*, governing these message exchanges form the interface specifications between network components. These protocols can significantly affect network security concerns, as will be seen later. In any event, because of the inability to make assumptions about the hardware and switches, one typically must expect malicious activity of several sorts.

1. Tapping of lines. While the relevant methods are beyond the scope of this discussion, it should be recognized that it is frequently a simple matter to record the message traffic passing through a given communications line, without detection by the participants in the communication [WEST70]. This problem is present whether the line is private, leased from a common carrier, or part of a broadcast satellite channel.

2. Introduction of spurious messages. It is often possible to introduce invalid messages with valid addresses into an operating network, in such a way that the injected messages pass all relevant consistency checks and are delivered as if the messages were genuine.

3. Retransmission of previously transmitted, valid messages. Given that it is possible both to record and introduce messages into a network, it is therefore possible to retransmit a copy of a previously transmitted message.

4. Disruption. It is possible that delivery of selected messages may be prevented: portions of messages may be altered, or complete blockage of communications paths may occur.

Each of the preceding threats can, in the absence of suitable safeguards, cause considerable damage to an operating network, and make it useless for communication. Tapping of lines leads to loss of privacy of the communicated information. Introduction of false messages makes reception of any message suspect. Even retransmission of an earlier message can cause considerable difficulty in some circumstances. Suppose the message is part of the sequence by which two parties communicate their identity to one another. Then it may be possible for some node to falsely identify itself in cases where the valid originator of the message was temporarily out of service.

While all of the previous examples require malicious intent, more and more applications of computer networks are becoming sensitive; increased motivation to disturb proper operation can be expected. Consider the attention that will be directed at such uses as military command and control systems (by which missile firing orders are sent) or commercial electronic funds transfer systems (with hundreds of billions of U.S. dollars worth of transactions daily).

1.2. Operational Assumptions

The following discussion of protection and security in computer networks will be based on several underlying assumptions.

1. Malicious attacks, including tapping, artificial message injection, and disruption are expected.

2. The insecure network provides the only available high bandwidth transmission paths between those sites which wish to communicate in a secure manner [1].

3. Reliable, private communication is desired.

4. A large number of separately protected logical channels are needed, even though they may be multiplexed on a much smaller number of physical channels.

5. High speed, inexpensive hardware encryption units are available.

It is believed that these assumptions correctly mirror many current and future environments. In the next sections, we outline properties encryption relevant to network use. Those interested in a deeper examination should see the companion papers in this issue. After this brief outline, the discussion of network security commences in earnest.

2. Encryption Algorithms and their Network Applications

Encryption provides a method of storing data in a form which is unintelligible without the "key" used in the encryption. Basically, encryption can be thought of as a mathematical function

$$E = F(D,K)$$

where D is data to be encoded, K is a key variable, and E is the resulting enciphered text. For F to be a useful function, there must exist an F', the inverse of F,

[1] It will turn out that *some* presumed secure and correct channel will be needed to get the secure data channel going, although the pre-existing secure channel can be awkward to use, with high delay and low bandwidth. Distribution of the priming information via armored truck might suffice, for example.

$$D = F'(E,K)$$

which, therefore, has the property that the original data can be recovered from the encrypted data if the value of the key variable originally used is known.

The use of F and F' is valuable only if it is impractical to recover D from E without knowledge of the corresponding K. A great deal of research has been done to develop algorithms which make it virtually impossible to do so, even given the availability of powerful computer tools.

The "strength" of an encryption algorithm is traditionally evaluated using the following assumptions. First, the algorithm is known to all involved. Second, the analyst has available to him a significant quantity of encrypted data and corresponding cleartext (i.e. the unencrypted text, also called plaintext). He may even have been able to cause messages of his choice to be encrypted. His task is to deduce, given an additional, unmatched piece of encrypted text, the corresponding cleartext. All of the matched text can be assumed to be encrypted through the use of the same key which was used to encrypt the unmatched segment. The difficulty of deducing the key is directly related to the strength of the algorithm.

F is invariably designed to mask statistical properties of the cleartext. The probability of each symbol of the encrypted character set appearing in an encoded message E ideally is to be equal. Further, the probability distribution of any pair (digram) of such characters is to be flat. Similarly, it is desirable that the n-gram probability distribution be as flat as possible, for each n. This characteristic is desired even in the face of skewed distributions in the cleartext, for it is the statistical structure of the input language, as it "shows through" to the encrypted language, which permits cryptanalysis.

The preceding characteristics, desirable from a protection viewpoint, have other implications. In particular, if any single bit of a cleartext message is altered, then the probability of any particular bit in the corresponding message being altered is approximately 1/2. Conversely, if any single bit in an encrypted message is changed, the probability is approximately 1/2 that any particular bit in the resulting decrypted message has been changed [FEIS75]. This property follows because of the necessity for flat n-gram distributions. As a result, encryption algorithms are excellent error detection mechanisms, as long as the recipient has any knowledge of the original cleartext transmission.

The strength of an encryption algorithm is also related to the ratio of the length of the key with the length of the data. Perfect ciphers, that completely mask statistical information, require keys of lengths equal to the data they encode. Fortunately, currently available algorithms are of such high quality that this ratio can be small; as a result, a key can be often reused for subsequent messages. That is, subsequent messages essentially extend the length of the data. It is still the case that keys need to be changed periodically to prevent the ratio from becoming too small, and thus the statistical information available to an analyst too great. The loss of protection which would result from a compromised key is thus also limited.

2.1. Public Key Encryption

Diffie and Hellman [DIFF76b] proposed a variation of conventional encryption methods that may, in some cases, have certain advantages over standard algorithms. In their class of algorithms, there exists

$$E = F(D,K)$$

as before to encode the data, and

$$D = F'(E,K')$$

to recover the data. The major difference is that the key K' used to decrypt the data is not equal to, and is impractical to derive from, the key K used to encode the data. Presumably there exists a pair generator which based on some input information, produces the matched keys K and K' with high strength (i.e., resistance to the derivation of K' given K, D, and matched $E = F(D,K)$).

Many public key algorithms have the property that either F or F' can be used for encryption, and both result in strong ciphers. That is, one can encode data using F', and decode using F. The Rivest et. al. algorithm is one that has this property [RIVE77a]. The property is useful both in key distribution and "digital signatures", and will be assumed here.

The potential value of such encryption algorithms lies in some expected simplifications in initial key distribution, since K can be publicly known; hence the name public key encryption. There are also simplifications for "digital signatures". These issues are examined further in sections 4 and 10. Rivest et. al., and Merkle and Hellman have proposed actual algorithms which are believed strong, but they have not yet been extensively evaluated [RIVE77a][HELL78].

Much of the remaining material in this survey is presented in a manner independent of whether conventional or public key based encryption is employed. Each case is considered separately when necessary.

2.2. *Error Detection, and Duplicate or Missing Blocks*

Given the general properties of encryption described in the earlier sections, it is a fairly simple matter to detect (but not correct) errors in encrypted messages. It is merely necessary that a small part of the message be redundant, and that the receiver know in advance the expected redundant part of the message. One can conclude that, in a block with k check bits, the probability of an undetected error upon receipt of the block is approximately $1/(2^k)$ for reasonable sized blocks, if the probabilistic assumption mentioned in section 2.0 is valid. For example, if three eight-bit characters are employed as checks, the probability of an undetected error is less than $1/(10^6)$.

In the case of natural language text, no special provisions need necessarily be made, since that text already contains considerable redundancy and casual inspection permits error detection with very high probability. The check field can also be combined with information required in the block for reasons other than encryption. In fact, the packet headers in most packet switched networks contain considerable, highly formatted information, which can serve the check function. For example, duplicate transmitted blocks may occur either because of a deliberate attempt or through abnormal operation of the network switching centers. To detect the duplication, it is customary to number each block in order of transmission. If this number contains enough bits, and the encryption block size matches the unit of transmission, the sequence number can serve as the check field.

Feistel et. al. [FEIS75] describe a variant of this method, called block chaining, in which a small segment of the preceding encrypted block is appended to the current cleartext block before encryption and transmission. The receiver can therefore easily check that blocks have been received in order, by making the obvious check, but if the check fails, he cannot tell how many blocks are missing. In both of these cases, once a block is lost, and not recoverable by lower level network protocols, some method for reestablishing validity is needed. One method is to obtain new matched keys. An alternative (essential for public key systems) is to employ an authentication protocol (as described in section 3) to choose a new, valid sequence number or data value to restart block chaining.

2.3. *Block vs. Stream Ciphers*

Another important characteristic of an encryption method, which both affects the strength of the algorithm and has implications for computer use, is whether it is a block or stream cipher. A stream cipher, in deciding how to encode the next bits of a message, can use the entire preceding portion of the message, as well as the key and the current bits. A block cipher, on the other hand, encodes each successive block of a message based only on that block and the given key. It is easier to construct strong stream ciphers than strong block ciphers. However, stream ciphers have the characteristic that an error in a given block makes subsequent blocks undecipherable. In many cases, either method may be satisfactory, since lower level network protocols can handle necessary retransmission of garbled or lost blocks. Independent of whether a block or stream cipher is employed, some check data, mentioned in section 2.2 above, is still required to detect invalid blocks. In the stream cipher case, when an invalid block is discovered after decoding, the decryption process must be reset to its state preceding the invalid block, so that the valid block, when received, will be properly handled.

On the other hand, stream ciphers are less acceptable for computer use in general. If one wishes to be able selectively to update portions of a long encrypted message (or file), then block ciphers permit decryption, update, and reencryption of the relevant blocks alone, while stream ciphers require reencryption of all subsequent blocks in the stream. Therefore, block ciphers are usually preferred. Fortunately, there exist reasonably strong block ciphers. The Lucifer system [FEIS73] is one such candidate. Whether or not the National Bureau of Standards' Data Encryption Standard (D.E.S.), with its 56 bit keys, is suitably strong is open to debate [DIFF77], but it is being accepted by many commercial users as adequate [NBS77].

2.4. *Network Applications of Encryption*

Four general uses of encryption have application in computer networks. Each is presented below.

1. *Authentication.* One of the important requirements in computer communications security is to provide a method by which participants of the communication can identify one another, in a secure manner. Encryption solves this problem in several ways. First, possession of the right key is taken as prima-facie evidence that the participant is able to engage in the message exchanges. The transmitter can be assured that only the holder of the key is able to send or receive transmissions in an intelligible way.

However, one is still subject to the problems caused by lost messages, replayed valid messages, and the reuse of keys for multiple conversations (which exascerbates the replay problem). A general authentication protocol which can detect receipt of previously recorded messages when the keys have not been changed is presented later. The actual procedures by which keys are distributed in the general case is of course important, and will be discussed in subsequent sections.

2. *Private Communication.* The traditional use for encryption has been in communications, where the sender and receiver do not trust the transmission medium, whether it be a hand carried note, or megabytes shipped over high capacity satellite channels. The use of encryption for protection of transmission becomes crucial in computer networks.

3. *Network Mail.* In the private communication function, it is generally understood that all parties wishing to communicate are present, and willing to tolerate some reasonable amoumt of overhead in order to get the private conversation established. Thus, a key distribution algorithm involving several messages and interaction with all participants would be acceptable. In the case of electronic mail, it

may be unreasonable for the actual transmission of what is frequently a short message to require such significant overhead. Further, mail should not require an active receiver, one that is actually logged in, at the time the message is received. On the other hand, some queueing delays at the sending or receiving site may be acceptable if the number of overhead messages can be significantly reduced.

4. *Digital Signatures.* The goal here is to allow the author of a digitally represented message to "sign" it in such a fashion that the "signature" has properties similar to an analog signature written in ink for the paper world. Without a suitable digital signature method, the growth of distributed systems may be seriously inhibited, since many transactions, such as those involved in banking, require a legally enforceable contract.

The properties desired of a digital signature method include the following:

1. Unforgeability: Only the actual author should be able to create the signature.

2. Authenticity: There must be a straightforward way to conclusively demonstrate the validity of a signature in case of dispute, even long after authorship.

3. No repudiation: It must not be possible for the author of signed correspondence to subsequently disclaim authorship.

4. Low cost and high convenience: The simpler and lower cost the method, the more likely it will be used.

Much of the remainder of this paper is devoted to discussion of how these general encryption applications can supported.

2.5. *Minimum Trusted Mechanism; Minimum Central Mechanism*

In all of the functions presented in section 2.4, it is desirable that there be minimum trusted mechanism involved [POPE74b]. This desire occurs because the more mechanism, the greater the opportunity for error, either by accident or by intention (perhaps by the developers or maintainers). One wishes to minimize the involvement of a central mechanism for analogous reasons. This fear of large, complex and central mechanisms is well justified, given the experience of failure of large central operating systems and data management systems to provide a reasonable level of protection against penetration [POPE74a][CARL75]. Kernel-based approaches to software architectures have been developed to address this problem, and have as their goal the minimization in size and complexity of central, trusted mechanisms. For more information about such designs, see [MCCA79][POPE79][DOWN79].

Some people are also distrustful that a centralized, governmental communication facility, or even a large common carrier, can assure privacy and other related characteristics. These general criteria are quite important to the safety and credibility of whatever system is eventually adopted. They also constrain the set of approaches that may be employed.

2.6. *Limitations of Encryption*

While encryption can contribute in useful ways to the protection of information in computing systems, there are a number of practical limitations to the class of applications for which it is viable. Several of these limitations are discussed below.

1. *Processing in Cleartext.* Most of the operations that one wishes to perform on data, from simple

arithmetic operations to the complex procedure of constructing indexes to data bases, require that the data be supplied in cleartext. Therefore, the internal controls of the operating system, and to some extent the applications software, must preserve protection controls while the cleartext data is present. While some have proposed that it might be possible to maintain the encrypted data in main memory, and have it decrypted only upon loading into cpu registers (and subsequently reencrypted before storage into memory), there are serious questions as to the feasibility of this approach [GAIN77]. The key management facility required is nontrivial, and the difficulties inherent in providing convenient controlled sharing seem forbidding. Another suggestion sometimes made is that an encoding algorithm be used which is homomorphic with respect to the desired operations [RIVE78]. Then the operation could be performed on the encrypted values, and the result can be decrypted as before. Unfortunately, those encoding schemes known with the necessary properties are not strong algorithms, nor is it generally believed that such methods can be constructed.

Therefore, since data must be processed in cleartext, other means are necessary for the protection of data from being compromised in the operating system by applications software. The remarks in the previous section concerning minimization of those additional means are very important in this context.

2. *Revocation.* Keys are similar to simple forms of *capabilities*, that have been proposed for operating systems [DENN66][DENN79]. They act as tickets and serve as conclusive evidence that the holder may access the corresponding data. Holders may pass keys, just as capabilities may be passed. Selective revocation of access is just as difficult as those known to simple capability methods [FABR74]. The only known method is to decrypt the data and reencrypt with a different key. This action invalidates all the old keys, and is obviously not very selective. Hence new keys must be redistributed to all those for whom access is still permitted.

3. *Protection Against Modification.* Encryption by itself provides no protection against inadvertent or intentional modification of the data. However, it can provide the means of detecting that modification. One need merely include as part of the encrypted data a number of check bits. When decryption is performed, if those bits do not match the expected values, then the data is invalid.

Detection of modification, however, is often not enough protection. In large data bases, for example, it is not uncommon for very long periods to elapse before any particular data item is referenced. It would be only at this point that a modification would be detected. While error correcting codes could be applied to the data after encryption to provide redundancy, these will not be helpful if a malicious user has succeeded in modifying stored data at will, as he can destroy the adjacent data into which the redundancy has been encoded. Therefore, very high quality recovery software would be necessary to restore the data from old archival records.

4. *Key Storage and Management.* Every data item that is to be protected independently of other data items requires encryption by its own key. This key must be stored as long as it is desired to be able to access the data. Thus, to be able to separately protect a large number of long lived data items, the key storage and management problem becomes formidable. The collection of keys immediately becomes so large that safe system storage is essential. After all, it is not practical to require a user to supply the key when needed, and it isn't even practical to embed the keys in applications software, since that would mean the applications software would require very high quality protection.

The problem of key storage is also present in the handling of removable media. Since an entire volume (tape or disk pack) can be encrypted with the same key (or small set of keys), the size of the problem is reduced. If archival media are encrypted, then the keys must be kept for a long period, in a highly reliable way. One solution to this problem would be to store the keys on the unit to which they correspond, perhaps even in several different places to avoid local errors on the medium. The keys would have to be protected, of course; a simple way would be to encrypt them with yet a different "master" key. The protection of this master key is absolutely essential to the system's security.

In addition, it is quite valuable for the access control decision to be dependent on the value of the data being protected, or even on the value of other, related data: salary fields are perhaps the most quoted example. In this case, the software involved, be it applications or system procedures, must maintain its own key table storage in order to successfully examine the cleartext form of the data. That storage, as well as the routines which directly access it, require a high quality protection mechanism beyond encryption.

Therefore, since a separate, reliable protection mechanism seems required for the heart of a multiuser system, it is not clear that the use of encryption (which requires the implementation of a second mechanism) is advisable for protection within the system. The system's protection mechanism can usually be straightforwardly extended to provide all necessary protection facilities.

3. System Authentication

Authentication refers to the identification of one member of a communication to the other, in a reliable, unforgeable way. In early interactive computer systems, the primary issue was to provide a method by which the operating system could determine the identity of the user who was attempting to log in. Typically, the user supplied confidential parameters, such as passwords or answers to personal questions, for checking at each login attempt. There was rarely any concern over the machine identifying itself to the user.

In networks, however, mutual authentication is of interest: each "end" of the channel may wish to assure itself of the identify of the other end. Quick inspection of the class of methods used in centralized systems shows that straightforward extensions of the methods are unacceptable. Suppose one required that each participant send a secret password to the other. Then the first member that sends the password is exposed. The other member may be an imposter, who has now received the necessary information to pose to other nodes in the network as the first member. Extension to a series of exchanges of secret information will not solve the problem. It only makes necessary a multi-step procedure by the imposter. A different approach is required.

There are a number of straightforward encryption-based authentication protocols which provide reliable mutual authentication without exposing either participant. The methods are robust in the face of all the network security threats mentioned earlier. The general principle involves the encryption of a rapidly changing unique value using a prearranged key, and has been independently rediscovered by a number of people [FEIS75][KENT76][POPE78]. An obvious application for such protocols is to establish a mutually agreed upon sequence number or block chaining initial value that can be used to authenticate communications over a secure channel whose keys have been used before. The sequence number or value should either be one that has not been used before, or should be selected at random, in order to protect against undetected replay of previous messages.

Below we outline a simple, general authentication sequence between nodes A and B. At the end of the sequence, A has reliably identified itself to B. A similar sequence is needed for B to identify itself to A. Typically, one expects to interleave the messages of both authentication sequences.

Assume that A uses a secret key, associated with itself, in the authentication sequence. The reliability of the authentication depends only on the security of that key. Assume that B holds A's matching key (as well as the matching keys for all other hosts to which B might talk).

1. B sends A, in cleartext, a random, unique data item, in this case the current time of day as known to B.

2. A encrypts the received time of day using its authentication key and sends the resulting ciphertext to B.

3. B decrypts A's authentication message, using A's matched key, and compares it with the time of day which B had sent. If they match, then B is satisfied that A was the originator of the message.

This simple protocol does not expose either A or B if the encryption algorithm is strong, since it should not be possible for a cryptanalyst to be able to deduce the key from the encrypted time of day, even knowing what the corresponding cleartext time of day was. Further, since the authentication messages change rapidly, it is not possible to record an old message and retransmit it.

To use such an authentication protocol to establish a sequence number or initial value for block chaining, it is merely necessary for A to include that information, before encryption, in its message to B in step 2 above.

4. Key Management

For several participants in a network conversation to communicate securely, it is necessary for them to obtain matching keys to encrypt and decrypt the transmitted data. It should be noted that a matched pair of keys forms a logical channel which is independent of all other such logical channels, but as real as any channel created by a network's transmission protocols. Possession of the key admits one to the channel. Without the key, the channel is unavailable. Since the common carrier function of the network is to provide many communication channels, how the keys which create the corresponding necessary private channels are supplied is obviously an important matter. The following sections describe various key distribution methods for both conventional and public key encryption systems.

4.1. Conventional Key Distribution

As there are, by assumption, no suitable transmission media for the keys other than the physical network, it will be necessary to devise means to distribute keys over the same physical channels by which actual data is transmitted. The safety of the logical channels over which the keys are to pass is crucial. Unfortunately, the only available method by which any data, including the keys, can be transmitted in a secure manner is through the very encryption whose initialization is at issue. This seeming circularity is actually easily broken through limited prior distribution of a small number of keys by secure means. The usual approach involves designating a host machine, or set of machines [HELL78], on the network to play the role of Key Distribution Center (KDC), at least for the desired connection. It is assumed that a pair of matched keys has been arranged previously between the KDC and each of the potential participants, say A1, A2, ..., Am. One of the participants, Ai, sends a short message to the KDC asking that matched key pairs be distributed to all the A's, including Ai. If the KDC's protection policy permits the connection, secure messages containing the key and other status information will be sent to each A, over the prearranged channels. Data can then be sent over the newly established logical channel. The prearranged key distribution channels carry a low quantity of traffic, and thus, recalling the discussion in section 2.0, the keys can be changed relatively infrequently by other means.

This general approach has many variations to support various desirable properties such as a distributed protection policy, integrity in face of crashes, and the like. Some of these are discussed below.

1. *Centralized Key Control.* Perhaps the simplest form of the key distribution method employs a single KDC for the entire network. Therefore n prearranged matched key pairs are required for a network with n distinguishable entities. An obvious disadvantage of this unadorned approach is its affect on network reliability. If communication with the KDC becomes impossible, either because the node on which the KDC is located is down, or because the network breaks, then the establishment of any further secure communication channels is impossible; if the overall system has been constructed to prevent any inter-user communication other than in a secure manner, then the entire network eventually stops. This design for distributed systems is, in general, unacceptable except when the underlying communications topology is a star and the KDC is located at the center. Note however that this drawback can be fairly easily remedied by the availability of redundant KDCs in case of failure of the main facility [1]. The redundant facility can be located at any site which supports a secure operating system and provides appropriate key generation facilities. Centralized key control can quite easily become a performance bottleneck however.

Needham and Schroeder present an example of how such a KDC would operate [NEED78]. Assume that A and B each have a secret key, Ka and Kb, known only to themselves and the KDC. To establish a connection, A sends a request to the KDC requesting a connection to B and includes an identifier (a random number perhaps). The KDC will send back to A i) a new key Kc to use in the connection, ii) the identifier, and iii) some information which A can send to B to establish the connection and prove A's identity. That message from the KDC to A is encrypted with A's secret key Ka. Thus, A is the only one who can receive it, and A knows that it is genuine. In addition, A can check the identifier to verify that it is not a replay of some previous request.

Once A has received this message, A sends to B the data from the KDC intended for B. That data includes the connection key Kc, as well as A's identity, all encrypted by B's secret key. Thus, B now knows the new key, that A is the other party, and that all this came from the KDC. However, B does not know that the message he just received is not a replay of some previous message. Thus, B must send an identifier to A encrypted by the connection key, upon which A can perform some function and return the result back to B. Now, B knows that A is current, i.e. there has not been a replay of previous messages. Figures 1 illustrates the messages involved. Of the five messages, two can be avoided in general by storing frequently used keys at the local sites, a technique known as caching.

2. *Fully Distributed Key Control.* Here it is possible for every "intelligent" node in the network to serve as a KDC for certain connections. (We assume some nodes are "dumb", such as terminals or possibly personal computers.) If the intended participants A1, A2, ..., Am reside at nodes N1, N2, ..., Nm, then only the KDCs at each of those nodes need be involved in the protection decision. One node chooses the key, and sends messages to each of the other KDCs. Each KDC can then decide whether the attempted channel is to be permitted, and reply to the originating KDC. At that point the keys would be distributed to the participants. This approach has the obvious advantage that the only nodes which must be properly functioning are those which support the intended participants. Each of the KDCs must be able to talk to all other KDCs in a secure manner, implying that $n^*(n-1)/2$ matched key

[1] The redundant KDCs form a simple distributed, replicated database, where the replicated information includes private keys and permission controls. However, the database is rarely updated, without serious requirements for synchronization among updates. It is not necessary for copies of a key at all sites to be updated simultaneously, for example. Therefore, little additional complexity from the distributed character of the key management function would be expected.

pairs must have been arranged. Of course, each node needs to store only n-1 of them. For such a method to be successful, it is also necessary for each KDC to talk with the participants at its own node in a secure fashion. This approach permits each host to enforce its own security policy if user software is forced by the local system architecture to use the network only through encrypted channels. This arrangement has appeal in decentralized organizations.

3. *Hierarchical Key Control.* This method distributes the key control function among "local", "regional", and "global" controllers. A local controller is able to communicate securely with entities in its immediate logical locale; that is, for those nodes with which matched key pairs have been arranged. If all the participants in a channel are within the same region, then the connection procedure is the same as for centralized control. If the participants belong to different regions, then it is necessary for the local controller of the originating participant to send a secure message to its regional controller, using a prearranged channel. The regional controller forwards the message to the appropriate local controller, who can communicate with the desired participant. Any of the three levels of KDCs can select the keys. The details of the protocol can vary at this point, depending on the exact manner in which the matched keys are distributed. This design approach obviously generalizes to multiple levels in the case of very large networks. It is analogous to national telephone exchanges, where the exchanges play a role very similar to the KDCs.

One of the desirable properties of this design is the limit it places on the combinatorics of key control. Each local KDC only has to prearrange channels for the potential participants in its area. Regional controllers only have to be able to communicate securely with local controllers. While the combinatorics of key control may not appear difficult enough to warrant this kind of solution, in certain circumstances the problem may be very serious, as discussed in the subsequent section on levels of integration.

The design also has a property not present in either of the preceding key control architectures: local consequences of local failures. If any component of the distributed key control facility should fail or be subverted, then only users local to the failed component are affected. Since the regional and global controllers are of considerable importance to the architecture, it would be advisable to replicate them, so that the crash of a single node will not segment the network.

All of these key control methods permit easy extension to the interconnection of different networks, with differing encryption disciplines. The usual way to connect different networks, with typically different transmission protocols, is to have a single host called a gateway common to both networks [CERF78] [BOGG80]. Inter-network data is sent to the gateway which forwards it toward the final destination. The gateway is responsible for any format conversions as well as the support of both systems' protocols and naming methods. If the networks' transmissions are encrypted in a manner similar to that described here, then the gateway might be responsible for decrypting the message and reencrypting it for retransmission in the next network. This step is necessary if the encryption algorithms differ, or if there are significant differences in protocol. If the facilities are compatible, then the gateway can merely serve as a regional key controller for both networks, or even be totally uninvolved.

There are strong similarities among these various methods of key distribution, and differences can be reduced further by designing hybrids to gain some of the advantages of each. Centralized control is a degenerate case of hierarchical control. Fully distributed control can be viewed as a variant of hierarchical control. Each host's KDC acts as a local key controller for that host's entities, and communicates with other local key controllers to accomplish a connection. In that case, of course, the communication is direct, without a regional controller required.

4.2. *Public Key Based Distribution Algorithms*

The class of public key algorithms discussed earlier have been suggested as candidates for key distribution methods that might be simpler than those described in the preceding sections. Recall that K'; the key used to decipher the encoded message, cannot be derived from K, the key used for encryption, or from matched encrypted and cleartext. Therefore, each user A, after obtaining a matched key pair <K, K'>, can publicize his key K. Another user B, wishing to send a message to A, can employ the publicly available key K. To reply, A employs B's public key. At first glance this mechanism seems to provide a simplified way to establish secure communication channels. No secure dialog with a key controller to initiate a channel appears necessary.

That is, an automated "telephone book" of public keys could generally be made available, and therefore whenever user A wishes to communicate with user B, A merely looks up B's public key in the book, encrypts the message with that key, and sends it to B [DIFF76b]. Therefore there is no key distribution problem at all. Further, no central authority is required initially to set up the channel between A and B.

It is clear, however, that this viewpoint is incorrect: some form of a central authority is needed and the protocol involved is no simpler nor any more efficient than one based on conventional algorithms [NEED78]. First, the safety of the public key scheme depends critically on the correct public key being selected by the sender. If the key listed with a name in the "telephone book" is the wrong one, then the protection supplied by public key encryption has been lost. Furthermore, maintenance of the (by necessity machine supported) book is nontrivial because keys will change, either because of the desire to replace a key which has been used for high amounts of data transmission, or because a key has been compromised through a variety of ways. There must be some source of carefully maintained "books" with the responsibility of carefully authenticating any changes and correctly sending out public keys (or entire copies of the book) upon request.

A modified version of Needham and Schroeder's proposal to deal with these issues is as follows. Assume that A and B each have a public key known to the authority, and a private key known only to themselves. Additionally, assume the authority has a public key known to all, and a private key known only to the authority.

A begins by sending to the authority a timestamped message requesting communication with B. The authority sends A the public key of B plus the timestamp, encrypted using the private key of the authority. A can decrypt this message using the public key of the authority, and is thus also sure of the source of the message. The timestamp guarantees that this is not an old message from the authority containing a key other than B's current public key [1].

A can now send messages to B because he knows B's public key. However, to identify himself, as well as to prevent a replay of previous transmissions, A now sends his name and an identifier to B, encrypted in B's public key. B now performs the first two steps above with the authority to retrieve A's public key. Then B sends to A the identifier just received, and an additional identifier, both encrypted with A's public key. A can decrypt that message and is now sure that he is talking to the current B. However, A must now send back the new identifier to B so that B can be sure he is talking to a current A. These messages are displayed in figure 2. The above protocol contains 7 messages, but 4 of them, those which retrieve the public keys, can be largely dispensed with by local caching of public keys. Thus, as in the conventional key distribution example, we again find that 3 messages are needed.

[1] These initial steps are essentially an adaptation of the authentication protocol given in section 3.

Some public key advocates have suggested ways other than caching in order to avoid requesting the public key from the central authority for each communication. Certificates is one such proposal [KOHN78]. A user can request that his public key be sent to him as a *certificate*, which is a user/public-key pair, together with some certifying information. For example, the user/public-key pair may be stored as a signed message[1] from the central authority. When the user wishes to communicate with other users, he sends the certificate to them. They each can check the validity of the certificate using the certifying information, and then retrieve the public key. Thus, the central authority is only needed once, when the initial certificate is requested.

Both certificates and caching have several problems. First, the mechanism used to store the cache of keys must be correct. Second, the user of the certificate must decode it and check it (verify the signature) each time before using it, or must also have a secure and correct way of storing the key. Perhaps most important, as keys change, the cache and old certificates become obsolete. This is essentially the capability revocation problem revisited [REDE74]. Either the keys must be verified (or re-requested) periodically, or a global search must be made whenever invalidating a key. Notice that even with the cache or certificates, an internal authentication mechanism is still required.

Public key systems also have the problem that it is more difficult to provide protection policy checks. In particular, conventional encryption mechanisms easily allow protection policy issues to be merged with key distribution. If two users may not communicate, then the key controller can refuse to distribute keys.[2] However, public key systems imply the knowledge of the public keys. Methods to add protection checks to public key systems add an additional layer of mechanism.

4.3. Comparison of Public and Conventional Key Distribution for Private Communication

It should be clear that both of the above protocols establish a secure channel, and that both require the same amount of overhead to establish a connection (3 messages). Even if that amount had been different by a message or two, the overhead is still small compared to the number of messages for which a typical connection will be used.

The above protocols can be modified to handle multiple authorities; such modifications have also been performed by Needham and Schroeder. Again, the number of messages can be reduced to three by caching.

It should also be noticed that the safety of these methods depends only on the safety of the secret keys in the conventional method, or the private keys in the public key method. Thus an equivalent amount of secure storage is required.

One might suspect, however, that the software required to implement a public key authority would be simpler than that for a KDC, and therefore easier to certify its correct operation. If this view were correct, it would make public key based encryption potentially superior to conventional algorithms, despite the equivalent protocol requirements. It is true that the contents of the authority need not be protected against unauthorized reference, since the public keys are to be available to all, while the keys used in the authentication protocol between the KDC and the user must be protected against reference. However, the standards of software reliability which need to be imposed on the authority for the sake of correctness are not substantially different from those required for the development of a secure KDC. More convincing, all of the KDC keys could be stored in encrypted

[1] See section 10 for a discussion of digital signatures.

[2] This approach blocks communication if the host operating systems are constructed in such a way as to prohibit cleartext communication over the network.

form, using a KDC master key, and only decrypted when needed. Then the security of the KDC is reduced to protection of the KDC's master key, and protection of the individual keys when in use. This situation is equivalent to the public key repository case, since there the private key of the repository must be safely stored and protected during use.

It has also been pointed out that a conventional KDC, since it issued the conversation key, can listen in, and in fact generate what appear to be valid messages. Such action cannot be done by the public key repository. This distinction is minor however. Given that both systems require a trusted agent, it is a simple matter to add a few lines of certified correct code to the conventional key agent (the KDC) that destroys conversation keys immediately after distribution. Thus the system characteristics of both conventional and public key algorithms are more similar than initially expected.

5. *Levels of Integration*

There are many possible choices of endpoints for the encryption channel in a computer network, each with their own tradeoffs. In a packet switched network, one could encrypt each line between two switches separately from all other lines. This is a low level choice, and is often called *link encryption.* Instead the endpoints of the encryption channels could be chosen at a higher architectural level: at the host machines which are connected to the network. Thus the encryption system would support host-host channels, and a message would be encrypted only once as it was sent through the network (or networks) rather than being decrypted and reencrypted a number of times, as implied by the low level choice. In fact, one could even choose a higher architectural level: endpoints could be individual processes within the operating systems of the machines that are attached to the network. If the user were employing an intelligent terminal, then the terminal is a candidate for an endpoint. This viewpoint envisions a single encryption channel from the user directly to the program with which he is interacting, even though that program might be running on a site other than the one to which the terminal is connected. This high level choice of endpoints is sometimes called *end-to-end encryption.*

The choice of architectural level in which the encryption is to be integrated has many ramifications. One of the most important is the combinatorics of key control versus the amount of trusted software.

In general, as one considers higher and higher system levels, the number of identifiable and separately protected entities in the system tends to increase, sometimes dramatically. For example, while there are less than a hundred hosts attached to the Arpanet [ROBE73], at a higher level there often are over a thousand processes concurrently operating, each one separately protected and controlled. The number of terminals is of course also high. This numerical increase means that the number of previously arranged secure channels — that is, the number of separately distributed matched key pairs — is correspondingly larger. Also, the rate at which keys must be generated and distributed can be dramatically increased.

In return for the additional cost and complexity which results from higher level choices, there can be significant reduction in the amount of software whose correct functioning must be assured. This issue is very important and must be carefully considered. It arises in the following way. When the lowest level (i.e. link encryption) is chosen, the data being communicated exists in cleartext form as it is passed from one encrypted link to the next by the switch. Therefore the software in the switch must be trusted not to intermix packets of different channels. If a higher level is selected, then protection errors in the switches are of little consequence. In a host-to-host level, however, operating system failures are still serious, because the data exists as cleartext while it is system resident.

In principle then, the highest level integration of encryption is most secure. However, it is still the case that the data must be maintained in clear form in the machine upon which processing is done. Therefore the more classical methods of protection within individual machines are still necessary, and the value of very high level end-end encryption may be somewhat lessened. A rather appealing choice of level that integrates effectively with kernel structured operating system architectures is outlined in the case study in section 8.

Another operational drawback to high level encryption should be pointed out. Once the data is encrypted, it is difficult to perform meaningful operations on it. Many front end systems provide such low level functions as packing, character erasures, and transmission on end-of-line or control-character detect. If the data is encrypted when it reaches the front end, then these functions cannot be performed. Any channel processing must be done above the level at which encryption takes place, despite the fact that performance and considerations such as the above sometimes imply a lower level.

6. Encryption Protocols

Network communication protocols concern the discipline imposed on messages sent throughout the network to control virtually all aspects of data traffic, both in amount and direction. Choice of protocol has dramatic impacts on the flexibility and bandwidth provided by the network. Since encryption facilities provide a potentially large set of logical channels, the protocols by which the operation of these channels is managed also has significant impact.

There are several important questions which any encryption protocol must answer:

1. How is the initial cleartext/ciphertext/cleartext channel from sender to receiver and back established?
2. How are cleartext addresses passed by the sender around the encryption facilities to the network without providing a path by which cleartext data can be inadvertently or intentionally leaked by the same means?
3. What facilities are provided for error recovery and resynchronization of the protocol?
4. How are channels closed?
5. How do the encryption protocols interact with the rest of the network protocols?
6. How much software is needed to implement the encryption protocols? Does the security of the network depend on this software?

One wishes a protocol which permits channels to be dynamically opened and closed, allows the traffic flow rate to be controlled (by the receiver presumably), provides reasonable error handling, and all with a minimum of mechanism upon which the security of the network depends. The more software involved, the more one must be concerned about the safety of the overall network. Performance resulting from use of the protocol must compare favorably with the attainable performance of the network using other suitable protocols without encryption. One would prefer a general protocol which could also be added to existing networks, disturbing their existing transmission mechanisms as little as possible. Each of these issues must be settled in addition to the level of integration of encryption, or the method of key distribution, which is selected.

Fortunately, the encryption channel can be managed independently of the conventional communication channel which is responsible for communication initiation and closing, flow control, error handling and the like. As a result, many protocol questions can be ignored by the encryption facilities, and handled by conventional means.

In section eight we outline a complete protocol in order to illustrate the ways in which these considerations interact and the independence which exists. The case considered employs distributed key distribution and an end-to-end architecture, all added to an existing network.

7. *Confinement*

To confine a program, process or user means that it will be unable to communicate at all other than through the explicitly controlled paths. Often improper communications are possible through subtle, sometimes timing dependent, channels. As an example, two processes might bypass the controlled channels by affecting each other's data throughput. Although many such improper channels are inherently error prone, the users may employ error detection and correction protocols to overcome that problem.

Unfortunately, the confinement problem in computer networks is particularly difficult to solve because most network designs require some information to be transmitted in cleartext form. This cleartext information, although limited, can be used for the passage of unauthorized information. In particular, the function of routing a message from computer to computer toward its final destination requires that the headers which contain network addresses and control information be in cleartext form, at least inside of the switching centers. A malicious user, cooperating with a penetrator, can send data by the ordering of messages among two communication channels. Even though the data of the communications is encrypted, the headers often are transmitted in cleartext form, unless link encryption is also used to encrypt the entire packet, including header. In any case, the routing task, often handled in large networks by a set of dedicated interconnected machines which form a subnet, requires host addresses in the clear within the switching machines. Thus a penetrator who can capture parts of the subnetwork can receive information. The only solutions to this problem appear to be certification of the secure nature of some parts of the subnetwork and host hardware/software. Work is in progress at the University of Texas on the application of program verification methods to this problem [GOOD77].

Certain confinement problems remain even if certification as suggested is applied. For example, the protocol implementing software in a given system usually simultaneously manipulates communications for several users. Either this software must be trusted, or data must be encrypted before it reaches this software. Even in this latter case, certain information may be passed between the user and the network software, and thus, potentially, to an unauthorized user. As an example, if a queue is used to hold information waiting to be sent from the user to the network, the user can receive information by noticing the amount drained from this queue by the network software. In almost any reasonable implementation on a system with finite resources, the user will at least be able to sense the time of data removal, if not the amount.

How well current program verification and certification methods apply here is open to question, since these confinement channels are quite likely to exist even in a correct implementation. That is, any feasible design seems to include such channels.

Given the difficulty of confinement enforcement, it is fortunate that most applications do not require it.

8. *Network Encryption Protocol Case Study: Private Communication at Process-Process Level*

It is useful to review an actual case study of how encryption was integrated into a real system to recognize the importance of the various issues outlined already. The example here was designed and implemented for the Arpanet, and is described in more detail by Popek and Kline [POPE78]; here we only outline the solution in general terms. The goal is to provide secure communication that does not involve application software in the security facilities, nor require trusting that software in order to be assured of the safety of the facilities. We also wish to minimize the amount of trusted system software.

The protocol provides process-to-process channels, and guarantees that it is not possible for application software running within the process to cause cleartext to be transmitted onto the network. Basic operation of the protocol is suggested in Figure 3. It is assumed, in keeping with the discussion in section 2.5, that the system software base at each node is a suitably small, secure operating system kernel, which operates correctly.

It is also expected that the amount of software involved in management of the network from the operating system's point of view is substantial, and therefore one wishes not to trust its correct operation.[1] Responsibilities of that software include establishing communications channels, supporting retransmission when errors are detected, controlling data flow rates, multiplexing multiple logical channels on the (usually) single physical network connection, and assisting or making routing decisions. We will call the modules which provide these functions the network manager (NM).

Let us assume for the moment that the keys have already been distributed, and logical channels established so far as the network managers are concerned. The operating system nucleus in each case has been augmented with new calls: *encrypt(channel name, data)* and *decrypt(channel name, data destination)*. Whenever a process wishes to send an encrypted block of data, it issues the encrypt call. The nucleus takes the data, causes it to be encrypted, and informs the network manager, who can read the block into its workspace. Assuming that the network manager knows what destination site is intended (which it must learn as part of establishing the logical channel), then it can place a cleartext header on the encrypted block and send it out onto the network. The cleartext header is essential, so that switching computers which typically make up a network can route the block appropriately.[2]

When the block arrives at the destination host computer, the network manager there reads it in and strips off the header. It then notifies the kernel, and tells the kernel for which process it claims the block is intended. The kernel informs the process, which can issue a decrypt call that causes the data to be decrypted with the key previously arranged for that process. If this block really was intended for this process (i.e., encrypted with the matching key), then the data is successfully received. Otherwise, decryption with the wrong key yields garbage. The encrypt and decrypt functions manage sequence numbers in a manner invisible to the user, as discussed in section 2.3.

Clearly this whole mechanism depends on suitable distribution of keys together with informing the network managers in a coordinated way of the appropriate endpoints of the channel. It is worth noting at this stage that matched keys form a well defined communication channel, and that in the structure just outlined, it is not possible for processes to communicate to the network or the network manager directly; only the encrypt and decrypt functions can be used for this purpose. It is for this latter reason that application software cannot communicate in cleartext over the network, an advantage

[1] As an example, in the Arpanet software for the Unix operating system, the network software is comparable in size to the operating system itself.

[2] Network encryption facilities must in general provide some way to supply the header of a message in cleartext, even though the body is encrypted.

if that code is not trusted (the usual assumption in military examples).

8.1. *Initial Connection*

To establish the secure channel, several steps are necessary. The local network manager must be informed to whom the local process wishes to communicate. This would be done by some highly constrained means. The network manager must communicate with the foreign network manager, and establish a name for this channel, as well as other state information such as flow control parameters. The network manager software involved need not be trusted. Once these steps are done, encryption keys need to be set up, in a safe way.

We first outline how this step would be done employing conventional encryption with fully distributed key management, and then comment on how it would change if public key systems were used.

Assume that there is a kernel-maintained key table which has entries of the form:

<foreign host name, channel name, sequence number, local process name, key>

There are also two additional kernel calls. *Open(foreign process name, local process name, channel name, policy-data)* makes the appropriate entry in the key table (if there isn't already one there for the given channel), setting the sequence number to an initial value, and sending a message to the foreign kernel of the form <local process name, channel name, policy-data, key>.[1]

If there already is an entry in the local key table, it should have been caused by the other host's kernel; so Open checks to make sure that the sequence number has been initialized, and doesn't generate a key, but sends out the same message, less the key. *Close(channel name)* deletes the indicated entry in the local key table, and sends a message to the foreign kernel to do the same.

The policy data supplied in the Open call, such as classification/clearance information, will be sent to the other site involved in the channel, so that it, too, will have the relevant basis to decide whether or not to allow this channel to be established.

Once both sides have issued corresponding Open calls, the processes can communicate. The following steps illustrate the overall sequence in more detail.

The host machines involved are numbered 1 and 2. Process A is at host 1 and B is at host 2. The channel name will be x.

1. A informs NM@1 (network manager at site 1), "connect using x to B@2". This message can be sent locally in the clear. If confinement between the network manager and local processes is important, other methods can be employed to limit the bandwidth between A and the NM.

2. NM@1 sends control messages to NM@2 including whatever host machine protocol messages are required.[2]

[1] The reader will note that the kernel to kernel message generated by the Open call must be sent securely, and therefore must employ a previously arranged key. The network manager must also be involved, since only it contains the software needed to manage the network.

[2] The host-host protocol messages would normally be sent encrypted using the NM-NM key in most implementations.

3. NM@2 receives an interrupt indicating normal message arrival, performs an I/O call to retrieve it, examines the header, determines that it is the recipient and processes the message.

4. NM@2 initiates step 2 at site 2, leading to step 3 being executed at site 1 in response. This exchange continues until NM@1 and NM@2 establish a logical channel, using x as their internal name for it.

5. NM@1 executes Open(B, A, x, policy-data).

6. In executing the Open, the kernel@1 generates or obtains a key, makes an entry in its Key Table, and sends a message over its secure channel to kernel@2, which makes a corresponding entry in its table and interrupts NM@2, giving it the triple <B,A,x>.

7. NM@2 issues the corresponding Open(A, B, x, policy-data'). This call interrupts B, and eventually causes the appropriate entry to be made in the kernel table at host 1. The making of that entry interrupts NM@1 and A@1.

8. A and B can now use the channel by issuing successive Encrypt and Decrypt calls.

There are a number of places in the mechanisms just described where failure can occur. If the network software in either of the hosts fails or decides not to open the channel, no kernel calls are involved and standard protocols operate. (If user notification is permitted, an additional confinement channel is present.) An Open may fail because the name x supplied was already in use, a protection policy check was not successful or because the kernel table was full. The caller is notified. He may try again. In the case of failure of an Open, it may be necessary for the kernel to execute most of the actions of Close to avoid race conditions that can result from other methods of indicating failure to the foreign site.

The encryption mechanism just outlined contains no error correction facilities. If messages are lost, or sequence numbers are out of order or duplicated, the kernel merely notifies the user and network software of the error and renders the channel unusable.[1] This action is taken on all channels, including the host-host protocol channels as well as the kernel-kernel channels. For every case but the last, Closes must be issued and a new channel created via Opens. In the last case, the procedures for bringing up the network must be used.

This simple-minded view is acceptable in part because the error rate which the logical encryption channel sees can be quite low. That is, the encryption channel is built on top of lower level facilities supplied by conventional network protocols, some implemented by the NM, which can handle transmission errors (forcing retransmission of errant blocks, for example) before they are visible to the encryption facilities. On highly error prone channels, additional protocol at the encryption level may still be necessary. See [KENT76] for a discussion of resynchronization of the sequencing supported by the encryption channel.

[1] Recall that these sequence numbers are added to the cleartext by the kernel Encrypt call before encryption. They are removed and checked after decryption by a Decrypt call issued at the receiving site, before delivery to the user. Hence, if desired, sequence numbers can be handled by the encryption unit itself, and never seen by kernel software. If such a choice is made, then the conventional network protocols supported by the NM will have to have another set of sequence numbers for error control.

From the protection viewpoint, one can consider the collection of NMs across the network as forming a single (distributed) domain. They may exchange information freely among them. No user process can send or receive data directly to or from an NM, except via narrow bandwidth channels through which control information is sent to the NM and status and error information is returned. These channels can be limited by adding parameterized calls to the kernel to pass the minimum amount of data to the NMs, and having the kernel post, as much as possible, status reports directly to the processes involved. The channel bandwidth cannot be zero, however.

The protocols just presented above in this case study can also be modified to use public key algorithms. The kernel, upon receiving the open request, should retrieve the public key of the recipient. Presumably, the kernel would employ a protocol with the authority to retrieve the public key, and then utilize the authentication mechanisms described in the protocols of section 3.

More precisely, in step 6 above, when the kernel receives the open call, it would retrieve the public key: either by looking it up in a cache, requesting it from the central authority, or via other methods such as certificates. Once the key is retrieved, the kernel would send a message to the other kernel, over the secure kernel-kernel channel, identifying the user and supplying those policy and authentication parameters required. The other kernel, upon receipt of that message, would retrieve the user's private key (from wherever local user private keys are stored) and continue the authentication sequence.

8.2. System Initialization Procedures

The task of bringing up the network software is composed of two important parts. First, it is necessary to establish keys for the secure kernel-kernel channels and the NM-NM channels. Next, the NM can initialize itself and its communications with other NMs. Finally, the kernel can initialize its communications with other kernels. This latter problem is essentially one of mutual authentication, of each kernel with the other member of the pair, and appropriate solutions depend upon the expected threats against which protection is desired.

The initialization of the kernel-kernel channel and NM-NM channel key table entries will require that the kernel maintain initial keys for this purpose. The kernel can not obtain these keys using the above mechanisms at initialization because they require the prior existence of the NM-NM and kernel-kernel channels. Thus, this circularity requires the kernel to maintain at least two key pairs.[1] However, such keys could be kept in read only memory of the encryption unit if desired.

The initialization of the NM-NM communications then proceeds as it would if encryption were not present. Once this NM-NM initialization is complete, the kernel-kernel connections could be established by the NM. At this point, the system would be ready for new connection establishment. It should be noted that, if desired, the kernels could then set up new keys for the kernel-kernel and NM-NM channels, thus only using the initialization keys for a short time. To avoid overhead at initialization time, and to limit the sizes of kernel key tables, NMs probably should only establish channels with other NMs when a user wants to connect to that particular foreign site, and perhaps close the NM-NM channel after all user channels are closed.

[1] In a centralized key controller version, the only keys which would be needed would be those for the channel between the key controller's NM and the host's NM, and the channel between the key controller's kernel and the host's kernel. In a distributed key management system, keys would be needed for each key manager.

This case study should serve to illustrate many of the issues present in the design of a suitable network encryption facility.

8.3. *Symmetry*

The case study portrayed a basically symmetric protocol suitable for use by intelligent nodes, a fairly general case. However, in some instances, one of the pair lacks algorithmic capacity, as illustrated by simple hardware terminals or simple microprocessors. Then a strongly asymmetric protocol is required, where the burden falls on the more powerful of the pair.

A form of this problem might also occur if encryption is not handled by the system, but rather by the user processes themselves. Then for certain operations, such as sending mail, the receiving user process might not even be present. (Note that such an approach may not guarantee the encryption of all network traffic.) The procedures outlined in the next section are oriented toward reducing the work of one of the members of the communicating pair.

9. *Network Mail*

Recall that network mail may often be short messages, to be delivered as soon as possible to the recipient site and stored there, even if the intended receiver is not currently logged in.

Assume that a user at one site wishes to send a message to a user at another site, but, because the second user may not be signed on at the time, a system process (sometimes called a "daemon") is used to receive the mail and deliver it to the user's "mailbox" file for his later inspection. It is desirable that the daemon process not require access to the cleartext form of the mail, for that would require the mail receiver mechanism to be trusted. This task can be accomplished by sending the mail to the daemon process in encrypted form and having the daemon put that encrypted data directly into the mailbox file. The user can decrypt it when he signs on to read his mail.

In either the conventional or public key case, the protocols described in section 4 can be employed with only slight modifications. In the conventional key case, the last two messages, which exchange an identifier to assure that the channel is current, must be dropped, since the recipient may not be present. After the sender requests and gets a conversation key (and a copy of it encrypted with the receiver's secret key), he appends the encrypted mail to the encrypted conversation key and sends both to the receiver. The receiving mail daemon can deliver the mail and key (both still encrypted), and the intended recipient can decrypt and read it at his leisure.

In the case of public keys, the sender retrieves the recipient's public key via an exchange with the repository, encrypts the mail, and sends it to the receiving site. Again the mail daemon delivers the encrypted mail, which can be read later by the recipient since he knows his private key. Again, the authentication part of the public key protocol must be dropped. In both of these approaches, since the authentication steps were not performed, the received mail may be a replay of a previous message. If detecting duplicate mail is important, the receiver must keep records of previous mail.

Both mechanisms outlined above do guarantee that only the desired recipient of a message will be able to read it. However, as pointed out, they don't guarantee to the recipient the identity of the sender. This problem is essentially that of digital signatures, and is discussed in section 10, next.

10. Digital Signatures

The need for digital signatures has by now become apparent. Applications such as bank transactions, military command and control orders, contract negotiation, will require reliable signatures. At first, it appeared that public key methods would be superior to conventional ones for use in digital message signatures. The method, assuming a suitable public key algorithm, is for the sender to encode the mail with his *private* key and then send it. The receiver decodes the message with the sender's *public* key. The usual view is that this procedure does not require a central authority, except to adjudicate an authorship challenge. However, two points should be noted. First, a central authority is needed by the recipient for aid in deciphering the first message received from any given author (to retrieve the corresponding public key, as mentioned in section 3.2). Second, the central authority must keep all old values of public keys in a reliable way to properly adjudicate conflicts over old signatures (consider the relevant lifetime of a signature on a real estate deed, for example).

Further, and more serious, the unadorned public key signature protocol just described has an important flaw. The author of signed messages can effectively disavow and repudiate his signatures at any time, merely by causing his secret key to be made public, or "compromised" [SALT78]. When such an event occurs, either by accident or intention, all messages previously "signed" using the given private key are invalidated, since the only proof of validity has been destroyed. Because the private key is now known, anyone could have created any message claimed to have been sent by the given author. None of the signatures can be relied upon.

Hence the validity of a signature on a message is only as safe as the *entire* future protection of the private key. Further, the ability to remove the protection resides in precisely the individual (the author) who should not hold that right. That is, one important purpose of a signature is to indicate responsibility for the content of the accompanying message in a way that cannot be later disavowed.

The situation with respect to signatures using conventional algorithms might initially appear slightly better. Rabin [RABI78] proposes a method of digital signatures based on any strong conventional algorithm. Like public key methods it too requires either a central authority, or an explicit agreement between the two parties involved, to get matters going.[1] Similarly, an adjudicator is required for challenges. Rabin's method however uses a large number of keys, with keys not being reused from message to message. As a result, if a few keys are compromised, other signatures based on other keys are still safe. However, that is not a real advantage over public key methods, since one could readily add a layer of protocol over the public key method to change keys for each message as Rabin does for conventional methods. One could even use a variant of Rabin's scheme itself with public keys, although it is easy to develop a simpler one.

However, all of the digital signature methods described or suggested above suffer from the problem of repudiation of signature via key compromise. Rabin's protocol or analogues to it merely limit the damage (or, equivalently, provide selectivity!). It appears that the problem is intrinsic to any approach in which the validity of an author's signature depends on secret information, which can potentially be revealed, either by the author or other interested parties. Surely improvement would be desirable.

[1] In his paper, Rabin describes an initialization method which involves an explicit contract between each pair of parties that wish to communicate with digitally signed messages. One can easily instead add a central authority to play this role, using suitable authentication protocols, thus obviating any need for two parties to make specific arrangements prior to exchanging signed correspondence.

10.1 Reliable Digital Signatures

A number of proposals have been made to augment or replace the unadorned approaches just outlined. One, suggested in [KLIN79] employs a network wide distributed signature facility. Others, based on analogues to notaries public in the paper world, or replicated, trusted archival facilities, provide a dependable timestamping mechanism so that authors cannot disavow earlier signed correspondence by causing their keys to be revealed.

10.2 Network Registry Based Signatures - A Conventional Key Approach

The registry solution is based on the obvious approach of interposing some trusted interpretive layer, a secure hardware and/or software "unit", between the author and his signature keys, whatever their form. Then it is a simple matter to organize the collection of units in the network to provide digital signature facilities. Consider all the cooperating units together as a distributed Network Registry (NR). Some secure communication protocol among the components of the Registry is required, but it can be very simple; low level link style encryption using conventional encryption would suffice.

Given that such facilities exist, then a simple implementation of digital signatures which does not require specialized protocols or encryption algorithms is as follows:

1. The author authenticates with a local component of the Network Registry (NR), creates a message, and hands the message to the NR together with the recipient identifier and an indication that a registered signature is desired.

2. The Network Registry (not necessarily the local component) computes a simple characteristic function of the message, author, recipient, and current time, encrypts the result with a key known only to the Network Registry, and forwards the resulting "signature block" to the recipient. The NR only retains the encryption key employed.

3. The recipient, when the message is received, can ask the NR if the message was indeed signed by the claimed author by presenting the signature block and message. Subsequent challenges are handled in the same way.

Certain precautions are needed to assure safety of the keys used to encrypt the signature blocks, including the use of different keys between pairs of distributed NR components, and a signature block computation which requires compromise of multiple components before signature validity is affected. For example, several NR components could each generate fragments of the keys being used. There is no need for all NR components even to be under control of a single centralized authority, so long as they can all cooperate.

10.3 Notary Public and Archive Based Solutions

Public key algorithms can provide safe signature methods also. One straightforward method is based on the behavior of notaries public in the paper world.[1] Briefly, there can be a number of independently operating (but perhaps licensed) *notary public machines* attached to the network. When a signed message has been produced, it can be sent to several of the notary public machines by the author, after the author has signed the message himself. The notary public machine timestamps the message, signs it himself (thereby encoding it a second time), and returns the result to the author. The author can then put the appropriate cleartext information around the doubly signed corresponder :e and send it to the intended receiver. He checks the notary's signature by decoding with the notary's public

[1] This approach was initially suggested to one of the authors by David Redell.

key, then decodes the message using the author's public key. Several notarized copies can be sent if desired, to increase safety.

The assumption underlying this method is that most of the notaries can be trusted. Since each notary timestamps its signature, it is not possible for the original author to disavow prior signed correspondence by "losing" his key at a given time. One might think however, that it is still possible for someone to claim that his key had been revealed without his knowledge sometime in the past, and selective messages forged. This problem can be guarded against by having each notary public return a copy of each notarized message to the author's permanent address. (This "patch" of course raises the question of how notaries are kept reliably informed of permanent addresses.)

Each notary is an independent facility, so that no coordination among them is required. Of course, if only one notary exists, then the approach is at best no improvement over the scheme presented in the previous section without multiple NR components. Danger of compromise of the notaries' private keys is reduced by the redundant facilities.

A related way to achieve reliable time registration of signed messages is for there to be a number of independent archival sites where either authors or recipients of signed mail may send copies of correspondence to be timestamped and stored permanently. Of course, the entire message need not be stored; just a characteristic function will do. Challenges are handled by interrogating the archives. The possibility of an individual's key being compromised and used without his knowledge can be treated in the same way as with notaries public.

10.4 Comparison of Signature Algorithms

The improved conventional key and public key based signature algorithms share many common characteristics. They each involve some generally trusted mechanism shared among all those communicating. The safety of signatures still depends on the future protection of keys as before, now those for the Network Registry, notaries public, or archive facilities. However, there are several crucial differences from previous proposals. First, the authors of messages do not retain the ability to repudiate signatures at will. Second, the new facilities can be structured so that failure or compromise of several of the components is necessary before signature validity is lost. In the early proposals, a single failure could lead to compromise.

11. User Authentication

While digital signatures are important, one must realize that there still *must* exist a guaranteed authentication mechanism by which an individual is authenticated to the system. Any reasonable communication system of course ultimately requires such a facility, for if one user can masquerade as another, all signature systems will fail. What is required is some reliable way to identify a user sitting at a terminal -- some method stronger than the password schemes used today. Perhaps an unforgeable mechanism based on fingerprints or other personal characteristics will emerge.

12. Conclusions

This discussion of network security has been intended to outline the issues in developing secure computer networks, as well as the context in which encryption algorithms will be increasingly used. It is surprising to note that, once the system implications are understood, public key algorithms and conventional algorithms are largely equivalent.

If one assumes that the purpose of a secure network is mainly to provide private pipes, similar to those supplied by common carriers, then general principles by which secure, common carrier based, point to point communication can be provided are reasonably well in hand. Of course, in any sophisticated implementation, there will surely be considerable careful engineering to be done. However, this conclusion rests on an important assumption that is not universally valid. The security, and correctness of function, of the underlying operating systems must be suitably high so that the network security methods described here are not being built on an unreliable base, obviating their safety. Fortunately, reasonably secure operating systems are well on their way, so that this intrinsic dependency of network security on appropriate operating system support should not seriously delay common carrier security [MCCA79][POPE78][FEIE79].

One could, however, take a rather different view of the nature of the network security problem: the goal might be to provide a high level extended machine for the user, in which no explicit awareness of the network is required. The underlying facility is trusted to securely move data from site to site as necessary to support whatever data types and operations are relevant to the user. The facility operates securely and with integrity in the face of unplanned crashes of any nodes in the network, synchronization of operations on user meaningful objects (such as Withdrawal from CheckingAccount) is reliably maintained, using minimum trusted mechanism. Other higher level security relevant operations beyond digital signatures are provided. If one takes such a high level view of the goal of network security, then the simple common carrier solutions are insufficient and more work remains.

13. Acknowledgments

The authors thank the referees for their comments. In particular, it is a pleasure to acknowledge Adele Goldberg for her help and guidance in revising the manuscript.

14. References

[AHO74] Aho, A., Hopcroft, J. and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, Mass., 1974.

[BOGG80] Boggs, D., Shoch, J., Taft, E., and R. Metcalfe, "Pup: An internetwork architecture", to appear in *IEEE Transactions on Computers*, Feb. 1980.

[BRAN73] Branstad, D. K., "Security aspects of computer networks," *AIAA Computer Network Systems Conference*, AIAA, April, 1973.

[BRAN75] Branstad, D. K., "Encryption protection in computer data communications," *Proceedings of the Fourth Data Communications Symposium*, 1975, 8-1 - 8-7.

[CARL75] Carlstedt, J., Bisbey, R. and G. Popek, *Pattern directed protection evaluation*, Report ISI/RR-75-31, University of Southern California, Information Sciences Institute, Marina Del Rey, California, 1975.

[CERF78] Cerf, V. and P. Kirstein, "Issues in packet-network interconnection," *Proceedings of the IEEE*, 66, 11 (Nov. 1978), 1386-1408.

[DENN66] Dennis, J. and E. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, 9, 3 (Mar. 1966), 143-155.

[DIFF76a] Diffie, W. and M. Hellman, "Multiuser cryptographic techniques," *Proceedings of the National Computer Conference*, 45 (1976), 109-112.

[MCCA79] McCauley, E. J. and P. J. Drongowski, "KSOS — The design of a secure operating system," *Proceedings of the National Computer Conference*, 48 (1979), 345-353.

[MERK78] Merkle, R., "Secure communication over insecure channels." *Communications of the ACM*, 21, 4 (Apr. 1978), 294-299.

[MEYE73] Meyer, C. H., "Design considerations for cryptography," *Proceedings of the National Computer Conference*, AFIPS, 42 (1973), 603-606.

[NBS77] National Bureau of Standards, "Data encryption standard", *Federal Information Processing Standards Publication 46*, 1977.

[NBS78a] National Bureau of Standards, "Design alternatives for computer network security," *National Bureau of Standards, Special Publication 500-21*, 1, 1978.

[NBS78b] National Bureau of Standards, "The network security center: A system level approach to computer security," *National Bureau of Standards, Special Publication 500-21*, 1, 1978.

[NEED78] Needham, R. and M. Schroeder, "Using encryption for authentication in large networks of computers", *Communications of the ACM*, 21, 12 (Dec 1978), 993-999.

[POPE74a] Popek, G. J., "Protection Structures," *IEEE Computer*, (Jul. 1974), 22-33.

[POPE74b] Popek, G. J., "A principle of kernel design," *Proceedings of the National Computer Conference*, 43 (1974), 977-978.

[POPE78] Popek, G. J. and C. S. Kline, "Design issues for secure computer networks", *Operating Systems, An Advanced Course*, R. Bayer, R. M. Graham, G. Seegmuller, ed., Springer-Verlag, 1978

[POPE79] Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A., Urban, M. and E. J. Walton, "UCLA Secure Unix," *Proceedings of the National Computer Conference*, 48 (1979), 355-364.

[RABI78] Rabin, M., "Digitalized Signatures", *Foundations of Secure Computing*, R. Demillo, et. al., editors, Academic Press, 1978.

[REDE74] Redell, D. D. and R. S. Fabry, "Selective revocation of capabilities", *Proceedings of the IRIA Conference on Protection in Operating Systems*, Rocquencourt, France, August, 1974.

[RIVE77a] Rivest, R. L., Shamir, A. and L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, M.I.T. Laboratory for Computer Science, Technical Memo LCS/TM82, Cambridge, Mass., April 4, 1977 (Revised August 31, 1977).

[RIVE77b] Rivest, R., private communications, 1977.

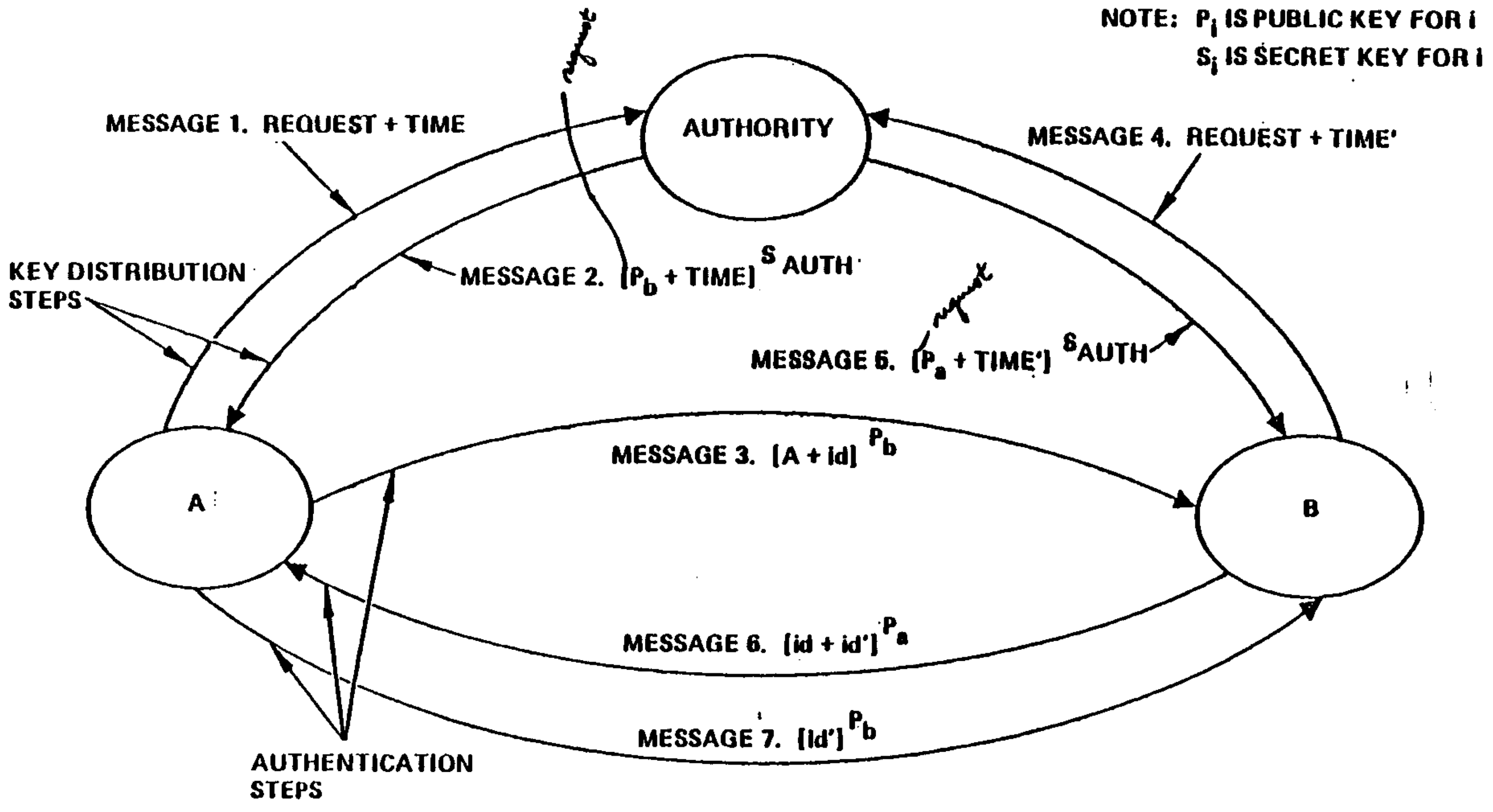
[RIVE78] Rivest, R. L., Shamir, A. and L. Adleman, "On digital signatures and public key cryptosystems," *Communications of the ACM*, 21, 2 (Feb 1978), 120-126.

[ROBE73] Roberts, L. and B. Wessler, "The ARPA Network," *Computer Communication Networks*, Prentice-Hall, 1973, 485-499.

[SALT78] Saltzer, J., "On digital signatures", *ACM Operating Systems Review*, 12, 2 (Apr. 1978), 2-14.

[SEND78] Sendrow, M., "Key management in EFT environments." *Proceedings of COMPCON*, 1978, 351-354.

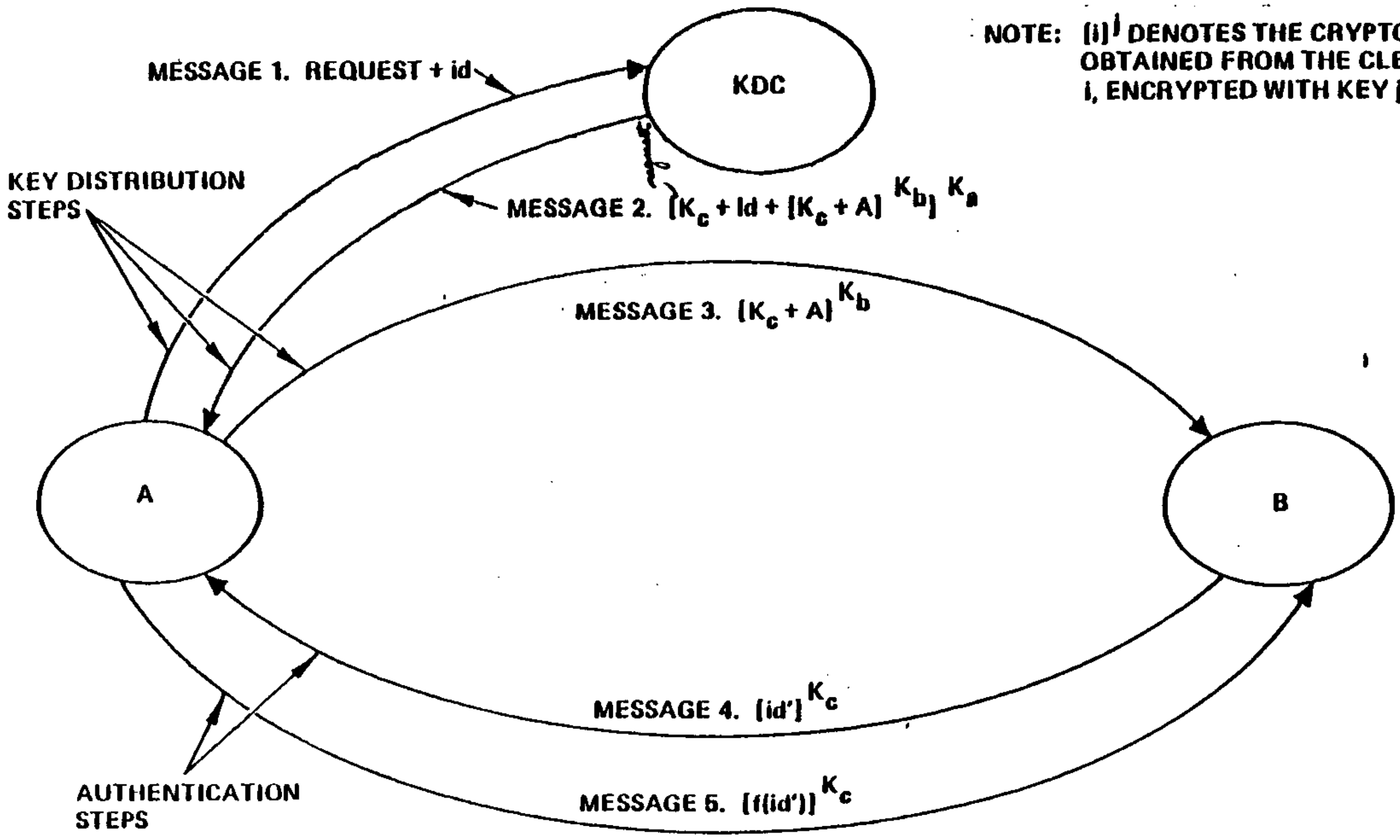
[WEST70] Westin, A. F., *Privacy and Freedom*, Atheneum Press, N.Y., 1970.



NOTE: P_i IS PUBLIC KEY FOR i
S_i IS SECRET KEY FOR i
AUTHORITY
A
B
MESSAGE 1. REQUEST + TIME
MESSAGE 2. $[P_b + \text{TIME}]^{S_{AUTH}}$
request
MESSAGE 3. $[A + id]^{P_b}$
MESSAGE 4. REQUEST + TIME'
MESSAGE 5. $[P_a + \text{TIME}']^{S_{AUTH}}$
request
MESSAGE 6. $[id + id']^{P_a}$
MESSAGE 7. $[id']^{P_b}$
KEY DISTRIBUTION STEPS
AUTHENTICATION STEPS

Figure 2. Key Distribution and Conversation Establishment: Public Key Algorithms

NOTE: $[i]^j$ DENOTES THE CRYPTOGRAM OBTAINED FROM THE CLEARTEXT i, ENCRYPTED WITH KEY j.



MESSAGE 1. REQUEST + id
KDC
KEY DISTRIBUTION STEPS
MESSAGE 2. $[K_c + \text{Id} + [K_c + A]^{K_b}]^{K_a}$
MESSAGE 3. $[K_c + A]^{K_b}$
A
B
MESSAGE 4. $[id']^{K_c}$
AUTHENTICATION STEPS
MESSAGE 5. $[f(id')]^{K_c}$

Figure 1. Key Distribution and Conversation Establishment: Conventional Key Algorithms

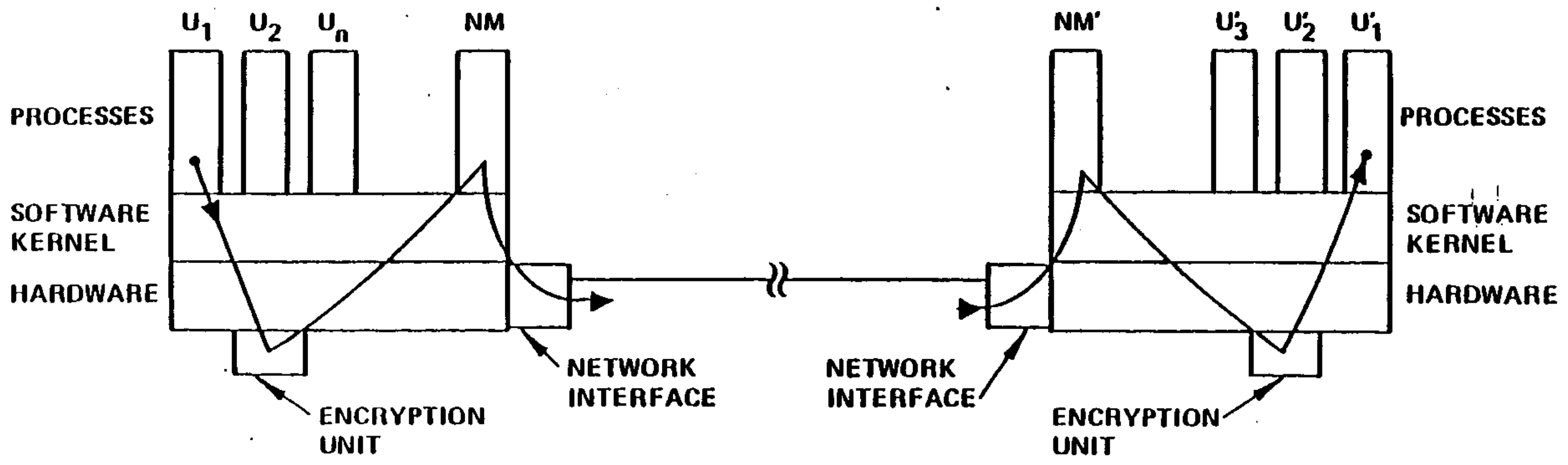


Figure 3. Data Flow in Process to Process Encrypted Channels.

6. *Technical Discussion: Distributed Systems*

Four reports are included in this section. The first presents a discussion of the central concept of *transparency* in computer networks and distributed systems. The second describes the transparent distributed file system in LOCUS, the prototype distributed system initially developed under this contract support. The last two papers concern reliability aspects of distributed systems: first replication of data and second the reconfiguration of the system in the face of failures of many kinds.

Transparency and its Limits in Distributed Operating Systems*

Gerald J. Popek and Bruce J. Walker

University of California at Los Angeles

Abstract

The continuing high cost of software has been increased in a number of ways by the advent of distributed systems. Building software for many purposes in a distributed environment is in a number of respects more difficult than for similar functions on a single machine, because of richer error modes, incompatible interfaces, and impoverished functionality.

Local area networks and the applications that are envisioned for them afford dramatic improvement to the software view of the distributed environment. This occurs because of i) the improved bandwidth/delay/reliability characteristics of typical local area networks over previously predominant, long haul communications; ii) the low cost of local net connections, which permit effective connection of small machines; and iii) the expected higher level of integration of the applications on the various machines in the local net compared to long haul practices.

In this paper, we discuss one of the major architectural approaches to ease software development in a distributed environment; *network transparency*. Relevant concepts and implementation issues are both discussed. Examples from existing systems are used for illustration. We suggest that similar concepts apply to long haul and low speed networks as well. Individual workstations connected by conventional links to a cluster of larger machines is an obvious example.

1 Introduction

It is well known that the cost of software development and maintenance is increasingly dominating the cost of hardware development and maintenance in computing systems today. Also, the connection of machines into various forms of distributed computing systems is an accelerating trend which is widely recognized to have just begun.

[1] This research was supported by the Advanced Research Projects Agency under research contract DSS-MDA-903-82-C-0189.

What is not as widely appreciated however, is that this second trend significantly exascerbates the first; that is, distributed software is generally far more difficult to develop, debug and maintain than single machine software. There are several reasons. First and foremost, in virtually all distributed systems which exist today, the way in which a user or a program accesses remote resources, be they data, programs, or tasks, is different from (and more complex than) local resource access. For example, a local file is typically accessed through *open*, *close*, *read* or *write* calls, while remote access may require execution of a file transfer protocol accompanied by renaming of that file. Second, the error modes present in distributed systems are richer and often more frequent than those present in single machine systems. For example, *partial failure* of a distributed computation can easily occur in a distributed environment, since one of the participating machines may fail while the others continue unaware. By contrast, in a single machine environment, both programmers and users generally expect that a failure will stop their entire computation. Third, in long haul networks, the bandwidth of the network is typically limited and the delay is considerable, especially when compared to analogous parameters within a machine connected to the network. Compare for example the 13 megabyte bus speed of a Vax-11/780 (or even the 5 megabyte Multibus) with 4800 baud international links, or even the 50 kilobit lines in the Arpanet. Further, when networks were being developed, the challenge was to find ways by which the machines involved could successfully exchange information, by whatever means, rather than finding an appropriate interface. The heterogeneity of the systems being connected didn't make the job any easier.

Local nets, by contrast, provide connections where the bandwidth and delay characteristics, while still considerably inferior to those within a mainframe, are dramatically better than the long haul case. The local net may be 3-10% of the internal bus of machines that are interconnected by it. One expects that since local networking is immature, further improvement is quite possible.

This bandwidth/delay improvement of between one and two orders of magnitude, accompanied by substantial cost reductions, permits one to seriously rethink the means by which systems are interconnected at the various software levels. One of the most significant results of such an effort is the concept of making the network of no concern, i.e. invisible, to most users and applications.

In the next sections, we examine the concept of such *transparency* and motivate its desirability. Then methods of realizing transparency are discussed and compared. Limits to transparency are pointed out, and their impact evaluated. Finally, conclusions regarding the proper role of transparency in local area, long haul, and low speed networks are offered.

2 The Concept of Transparency

We have already pointed out the unpleasant truth of distributed systems; software in that environment, especially for true distributed applications, is often far harder to design, implement, debug, and maintain than the analogous software written for a centralized system. The reasons - the errors/failures problem and issues of heterogeneity have also already been mentioned.

Many of these problems need not be intrinsic to the application view of distributed systems, however. It may be perfectly reasonable to open a file in precisely the same manner independent of whether the file is local or remote; i.e. issue the same system call, with the same parameters in the same order, etc. That is, the syntax and semantics of services should not be affected by whether or not a given function involves local or remote support. If *open (file-name)* is used to access local files, it also is used to access remote files. That is, the network becomes "invisible", analogous to the way that virtual memory hides secondary store. Hiding the existence of the network, at least so far as the nature of interfaces is concerned, can greatly ease software development.

This solution is called *network transparency*; all resources are accessed in the same manner independent of their location.

Of course, one still needs some way to control resource location for optimization purposes, but that control should be separated from the syntax and semantics of the system calls used to *access* the resources. That is, the existence of the network should not concern the user or application programs in the way that resources are accessed. Ideally then, one would like the graceful behavior of an integrated storage and processing system for the entire network while still retaining the many advantages of the distributed system architecture. If such a goal could be achieved, its advantages include the following.

1. Easier software development. Since there is only one way to access resources, and the details of moving data across the network are built in, individual software packages do not require special purpose software for this purpose. Functions are location independent.[1]

2. Incremental Change Supported. Changes made below the level of the network wide storage system are not visible to application software. Therefore, changes in resource support can be made more easily.

[1] On the local network installed at UCLA, the first version of network software made the network appear like another Arpanet, in which the details of the network are visible to application programs. The construction of a network-wide printer daemon required over 20 processes and several thousand lines of code beyond the spooler function itself, to deal with error conditions, size problems, and asynchronous events. Once a network transparent system was installed, virtually all of this mechanism vanished.

3. Potential for Increased Reliability. Local networks, with a fair level of redundancy of resources (both hardware and stored data), possess considerable potential for reliable, available operation. However, if this potential is to be realized, it must be possible to easily substitute various resources for one another (including processors, copies of files and programs, etc.). A uniform interface which hides the binding of those resources to programs would seem to be necessary if the higher reliability goal is to be realized.

4. Simpler User Model. By taking care of the details of managing the network, the user sees a conceptually simpler storage facility, composed merely of files, without machine boundaries, replicated copies, etc. The same is true for other user visible resources. Therefore, when moving from a simple machine to multisite operation, the user view is not needlessly disturbed.

In the next sections, we outline principles of network transparency and associated design issues that arise. Subsequently, we point out the full transparency is not achievable, or even desirable, in practice. Nevertheless, exceptions must be made with great care, in order not to destroy the benefits transparency is designed (and able) to provide.

3 Dimensions to Transparency

There are a number of aspects to network transparency. First is the manner in which objects and resources are named. Clearly, each object (such as a file) must have a globally unique name from the application point of view. In particular, the meaning of a name, i.e. the object with which it is associated, should not depend on the site in the network from which it is issued. This characteristic is called *name transparency*. Without it, moving or distributing software in the network can be very painful, since the effect of the program changes with the move.

Second, is the location of the resource encoded in the name? This is often the approach taken in early systems; the site name would be prepended to the existing file name to provide uniqueness. However, this choice has the unfortunate effect of making it quite difficult to move a file. The reason is that it is common to embed file names in programs. Moving a file implies changing its name, making previously correct software no longer operational. The challenge, of course, is to provide *location transparency* in an efficient way, without significant system overhead to find the node(s) storing the file. Some additional mechanism is necessary, since location would no longer be discernible from inspection of the name of the object.

In addition, automatically replicated storage is one important way that a system can increase effective reliability and availability to the user. To do so transparently, it is essential that the location of an object not be reflected in the name, since then it would be rather difficult to store a copy of the object at more than one site, and have any copy accessed automatically when others were not available.

Semantic consistency is a third, important issue. By this we mean that system services, support libraries, commonly used application programs, and the like have the same effect independent of the site on which they are executed. This property is obviously essential if one is to be able to move program execution and data storage sites; a critical facility for the sake of reliability and availability methods. However, it is also very important from the viewpoint of managing software maintenance. If multiple versions of software are needed to compensate for subtle differences in environments, the maintenance problem grows significantly.

Actually, this environment consistency problem is an old one. People commonly complain that it is not possible to directly transport software even between two identical hardware environments, because of local changes to the operating systems, because of differences in location or naming of libraries and other application services, or even because of differences between the versions of software currently installed. While this compatibility issue is serious enough among unconnected installations, it is far worse in a distributed system, where a much more intimate mode of cooperation among sites is likely to be the rule.

Unfortunately, this issue of semantic consistency conflicts with goals of local autonomy, because it constrains the conditions under which individual sites install new software and customize existing facilities. Methods by which this conflict can be reconciled are outlined later.

4 Transparency and System Levels

There are many levels within a distributed system at which one could choose to provide transparency. One could build a true distributed operating system, for example, in which the existence of the network is largely hidden near the device driver level in the operating system nucleus. Alternately, one could construct a layer of software over the operating systems but below application software. That layer would be responsible for dealing with distribution issues and functions. These first two approaches probably look very similar to users. Or, one could instead provide transparency via an extended programming language; those who wrote in that language would enjoy a network-wide virtual programming environment. Lastly, one might build transparency into an important application subsystem, such as a database. All of these methods have been or are being pursued. Below, we briefly comment on various of these approaches.

4.1 Distributed Database Supported Transparency

Most developers of distributed database systems mean by that label that the user of the distributed database system sees what appears to be a system-wide transparent database. Queries are expressed in the same manner as when all the data is resident on a single machine. The database system is responsible for keeping track of where the data is, and processing the query, including any optimization which may be appropriate. A distributed data base can be built on top of multiple copies of identical single machine operating systems, or it may even be constructed on top of a set of differing operating systems. The latter case may imply multiple implementations, of course. In some cases one may put an additional layer on top of existing heterogeneous database systems, to provide yet again another, albeit common, interface. Each of these approaches have been followed in practice. Distributed Ingres [STON 76] is following the first, R* will support the second [LIND 80a], and the third is being pursued in current research.

4.2 Programming Language Supported Transparency

As an alternative approach to transparency, one could provide distribution mechanisms in a suitable programming language. Argus [LISK 81] is an example of such an approach. In addition to providing primitives in the language to facilitate the construction of distributed programs that actually use the available parallelism, the language compiler and runtime system are responsible for providing many of the services that will be outlined in this paper.

In particular, some researchers have proposed some form of a *remote procedure call* as the central approach to transparency in distributed systems [NELS 81] [SPEC 82] [LISK 81]. In that view, a procedure call should have the same semantics (and syntax) whether the called procedure is to be executed locally or remotely. Sometimes restrictions on the remote case are imposed, such as forbidding the sharing of global variables among procedures that can be executed on differing sites. With this restriction, all data exchanged between the calling and called procedure is passed as explicit arguments. Specter shows that, with special protocols and microcode, even shared globals can often be supported in a local area network without serious performance degradation. Nevertheless, providing suitable semantics for remote procedure calls is a difficult task in the face of failures. Argus treats each call as a complete transaction. Nelson [NELS 81] gives a series of protocols intended to correct matters when errors occur.

4.3 The Operating System Level

In our judgment, however, if it is feasible within the constraints of existing systems, or if such constraints do not exist, it is more attractive to provide transparency at the operating system level than only via the alternatives mentioned above. In this way, all clients, including the database and language processor, can capitalize on the supporting facilities, customizing them as appropriate. Much of the work required to provide transparency is the same, independent of the level at which it is done. A global name map mechanism is needed. Inter-task communication across the network must be supported in a standardized way. Distributed recovery is important. Given that there is substantial work to be done, it is highly desirable that the results of that work be available to as many clients as possible. Putting transparency only into the dbms or the language means that users not accessing the database, or programs not written in the extended language, do not have access to these desirable facilities. This situation is especially regrettable when *part* of a given function is built using the transparent facilities, but the rest cannot be, because it has been written in a different language, or must use other than the dbms or language functions.

For these reasons, in those few cases in this paper where it is necessary to assume a context, we will couch our discussion in operating systems terms. The only significant case where this view is apparent is in the naming discussion, where we assume that the underlying structure is an extended *directory system*, rather than some other representation of the name-to-location mapping.

5 Optimization Control

One of the functions that conventional, non-transparent systems provide is the ability of applications to take explicit network related actions for the sake of optimization. Placing a given file at a particular site, or moving a resource from one site to another, are obvious examples. In a distributed environment, one in general has the choice of moving the data to the process or the process to the data. When location of resources are explicitly part of the application interface, then it is generally clear how to take site dependent actions. However, when a high level of transparency is provided, location of resources is by definition not apparent to the program. Unless some means are provided in a transparent environment to accomplish the same optimization actions, one could fairly expect significant performance difficulties, even in a local network.

The principle by which this optimization goal can be accomplished is straightforward. One can think of the set of functions by which the distributed system provides service to applications as an *effect language*. This language in the typical case is composed of a set of system calls; in some cases a more extensive JCL is also supplied as part of the system interface. We argue that a separate *optimization language* should be created, "orthogonal" to the effect language. The optimization language is *semantics free*, in the sense that whatever is stated in the optimization language cannot affect the outcome of a program: i.e. cannot change the result of any statement or series of statements in the effect language. The optimization language permits one to determine the location of resources, request that the system move a resource, etc. Since in a transparent system such actions do not affect the manner by which resources are accessed by applications, it is straightforward to make the optimization language free of semantic effect. In the case of a distributed operating system, this orthogonal language might consist of additional system calls such as:

> ***my_loc*** returns node number at which call is made;
> ***object_loc (object_name);*** returns node number of location of named object
> ***make_near (object_name, site_name);*** tries to move ***object_name*** to a site where access from site ***site_name*** will be efficient.

With this kind of interface, a program such as the fragment below executes correctly whether or not the optimization commands are successfully performed by the system.

```
.
.
x <- my_loc
make_near (file_foo, x);
open (file_foo)
.
.
```

The same advantages accrue to operations such as inter-process communication, system management, etc.

Now note that the total set of system calls are available to a given program, so that it is certainly possible to create a program which will behave differently depending on the site of execution (i.e. not in a transparent manner). Consider the following program fragment:

```
if object_loc (file_foo) = my_loc
    then call x else call y
```

We consider the above a necessary dimension to the transparency concept.

This general view of orthogonal languages applies equally well to programming language based approaches to distributed systems. The effect language is the normal procedure language, as extended for example to provide tools for parallel execution. The optimization language represents an additional set of extensions that are non-procedural, as declarations are, but which do not affect the meaning of the program.

Another way of viewing this semantics-free optimization language is that if the system were to ignore the commands, the program would still work correctly. In fact the system might do just that if, for example, the application asked for a file to be made local, but there was not enough room on the local storage medium. Lastly, as indicated above, if the application really wishes not to run unless its optimization commands were successful, it should be straightforward to do so.

6 Naming

As we saw earlier, naming is a key component of transparency. It is desirable that each resource (data, processing agent or device) be uniquely identified (name transparency) so that uttering a given name always references the same resource. This form of transparency allows software to migrate both before and during it's execution. As we shall see in this section however, name transparency can be misleading, even in the single site situation, because name-to-resource translation is often context dependent.

Location transparency is a further refinement to name transparency. To attain location transparency one must avoid having resource location be part of the resource name. By so doing, one is given the flexibility of transparently a) moving resources from site to site (akin to the flexibility data independence yields in transparently restructuring database data); and b) substituting one copy of a resource for another.

In this section we will investigate not only name and location transparency but also the whole area of contexts, some performance considerations, the role of resource replication and some local autonomy considerations.

6.1 Single Site Global Naming

Before considering naming in a multisite environment, consider the single site. One would imagine that most single site environments have unique resource naming facilities. Nonetheless, the action of a given command may often be context dependent, which is to say that the identical sequence of commands may not always have the same effect. Such mechanisms are not new. The *working directory* facility in most tree structures filesystems is perhaps the best example. Each process, by setting it's working directory, changes the effect of names uttered by programs run in the process. It provides a way of expanding abbreviated pathnames to globally unique names.

Another form of context is the use of aliasing. A user or process may have a translation table that is used in preprocessing names. Different users, then, can access different resources using the same name. The syntatic context in which a name is uttered can also be significant. Typing the word "who" in the context of a command may cause some extensive searching to find the load module. On the other hand, the word "who" in the context of the command *copy who to who.old* will not invoke the extensive searching but may only look for something called 'who' in the working directory.

So we see that eventhough global naming provides a means for uniquely identifying resources in a single site system, there are several ways the context in which a name is uttered can influence which resource is accessed. Next we turn to the issue of contexts in a distributed environment.

6.2 Naming Contexts

Much of the needed mechanism to support transparency is concerned with name translation. Similarly, many of the ostensible problems with transparency also concern the way that a name is associated with an object. The distributed directory system alluded to in this paper provides a system-wide decision about the association between object names and the objects themselves. A number of transparency problems can be avoided if it were possible to associate with a process or user a different set of decisions. We will call the preprocessing of names according to a set of per user or process rules the application of a naming *context*.

Use of contexts can solve a set of problems in a network transparent distributed system. Consider the following problems:

1. Two network transparent systems that were developed independently using the same distributed operating system are subsequently merged. Clearly one must expect numbers of name conflicts; i.e. object x exists in both networks and refers to different files.

2. A set of programs are imported from another environment; there different naming conventions were used so that changes are needed if the programs are to be run in the target distributed system. Source code may not be available. While this problems occurs today in single systems, one expects it to be more frequent in a large scale transparent system with a single fixed naming hierarchy.

Both of these examples, as well as the temporary file problem mentioned below, have the characteristic that the global, hierarchical name structure provided by the distributed system did not meet the needs of individual programs.

Suppose by contrast it were possible to construct an efficient name map package on a per program basis that is invoked whenever a name is issued. That package would convert the program issued names to appropriate global system names. The database used by that package is the naming context within which the program is then run. There are many degrees of generality one might achieve with a context mechanism, from the simple working directory mentioned earlier, to IBM's JCL, to Unix shell aliases, to the extensive *closure* mechanisms discussed by Saltzer [SALT 78].

Here we are interested in those aspects of such proposals which aid in solution of distributed system problems like those mentioned above. What is needed is the ability to replace a partial string of components of a path name with a different string of components. For example, in the case of merging two networks, a process might issue the name /bin/special.program, and the context mechanism might convert that name to /network1/bin/special.program, where /network1 is a network-wide directory which contains special programs needed for software developed in network1 that use library file names incompatible with naming conventions in network2. One might imagine then an implementation of a context mechanism as a hash table, where the string to be hashed is part of a path name, and the entry to be found is the path name part to be substituted. Definition of contexts should be loadable and changeable under program control. Presumably contexts are stored in files in the global directory system. A default context can be invoked at user login time.

6.3 Local Data

There are numbers of situations where, for the sake of significantly improved performance, or to ease operational problems, it may be desirable to make a few exceptions to complete transparency. In Unix, for example, it is conventional for the /tmp directory to be used for temporary storage needed by running programs. Typically, any user may read or write into that directory, creating files as necessary. Programs have path names embedded into their code which generate and use files in /tmp. It is rare that any files in /tmp are shared among disparate users. The question arises: should /tmp be treated as just another globally known directory in the network-wide naming system? If so, there will be the unavoidable overheads of synchronizing access to files in that directory, maintaining consistency among multiple copies if they exist, etc. Is the /tmp *directory* itself highly replicated? If so, these overheads can be significant. If not, then when most sites create a temporary file, that creation will involve a remote directory update. These costs will be frequently paid, and generally for no good reason.

Alternately, one could introduce the concept of *local file directories*. A local file directory is not network transparent. A directory of that name exists on each node, and files created in that directory are accessible only from that node. /tmp/foo on site i and /tmp/foo on site j are different files. Since this local file directory is not replicated and not globally known, access can be faster and inexpensive. Berkeley's COCANET effectively uses this solution. One could even give each local directory two names. One name is the common shared name; /tmp in the above. Each such directory would also have a global, network transparent name as well; /tmpi and /tmpj for example, so that if necessary, temporary files could be remotely accessed in a transparent fashion.

Such a compromise induces serious problems however, and should be generally avoided. It makes process transparency nearly impossible to achieve, for example. Two programs which run successfully on the same site, exchanging data through temporary files, will not run if separated. There are many better solutions to the desire to avoid overhead in the support of temporary files that do not involve compromising transparency. For example, the context mechanism discussed below can be used to cause temporary file names issued by a program to be mapped to a (globally available, network transparent) directory which happens to be locally or near locally stored, and not replicated.

The desire for site dependent naming is also raised by system initialization issues. Once the basic system is booted on a node, a program which finishes initialization work may wish to open a file which contains local configuration parameters. That initialization program wishes to be simple. If the initialization directory is a local directory, then the identical copy of the initialization program could be run on each site with the appropriate effect. This motivation is especially strong if an existing, single site program is being adapted for use in the distributed environment. A better solution is to add to the system a call that permits a program to determine what site it is running on, for then the calling program can compose a specific (global, network transparent) file name from that information. Such a mechanism can be used to get the desired effect.

7 Heterogeneity

Despite the clear and overriding advantages to transparency, there are several intrinsic limitations in certain environments to making the network system entirely transparent. The problems are presented by heterogeneity of hardware and software interfaces. Additional problems that are often mentioned, such as local autonomy, dynamic optimization desires, or the necessity to support slow links connecting subnets themselves connected by local area technology, are of far less importance in our judgement. Each of these issues is discussed below.

7.1 Hardware Heterogeneity

Differences in hardware among the machines connected to the network fall into several major categories. First, the instruction sets of the machines may differ. This means that a load module for one machine cannot be executed on another, incompatible machine. A distributed system which required the user (or his program) to issue one name to get a function done when executing on one machine, but another name for the same function when execution is to occur on another machine, is not transparent. What one really wants in this case is for a single application visible name to be mapped to multiple objects, with the choice of mapping and execution site (including shipping of needed data) to be automatically done in a way that depends on external conditions. However, this one-to-many mapping must be set up carefully, as there are other times when the individual objects need to be named individually (when a load module is to be replaced, for example).

Second, while differences in instruction set can be hidden via methods inspired by the above considerations, incompatibilities in data representations are more difficult to handle.[RASH 81] Suppose, for example, that program x, for which there exists a load module only for machine type X, wishes to operate on data produced by program y, which runs only on machine type Y. Unfortunately, the machines involved are Vaxes and M68000s, and the data includes character arrays. The M68000, with 16 bit words, is byte addressable, and if the low order byte of a word has address i, then the high order byte has address i+1. The Vax, also byte addressible, addresses bytes within a word in the reverse order. Since both instruction sets enforce the corresponding conventions, attempts to index through the character array will have different effects on the two machines. As a result, it is not possible in general to provide transparent access to that data. This is one example of the "big ender/little ender" problem described by Cohen. Differences in floating point formats are another well known example, which may be diminished by impending standards. Incompatibilities in higher level data structures are usually not an intrinsic problem, as those are generally induced by compiler conventions, which can be altered to conform to a common representation. Nevertheless, reaching such an agreement can be rather difficult, and there is always the remaining problem of compatibility with the past.

The third dimension of hardware incompatibility concerns configurations. While the cpus may be functionally identical, one system may have more space, either in main or secondary store, or may have certain attached peripherals essential to execution. This problem is generally relatively minor. The availability of virtual memory on most machines, even microprocessor chips, relieves the main store problem, and a good distributed file system can hide secondary store boundaries. The other problems can largely be treated in a similar way to the incompatible instruction set issue, that is, by automatically and invisibly causing the invoked program to be run on a suitable configuration if one exists in the network.

Hence, we conclude that hardware heterogeneity, with the exception of data representation problems, presents no significant obstacle to complete transparency.

7.2 Operating System Heterogeneity

While hardware issues can largely be handled by suitable techniques, if there are requirements to support more than one operating system interface in the transparent distributed system, serious problems can arise. The issue is *not* the heterogeneity of the system call interface of and by itself. After all, many systems today support more than one such interface. Usually, one set of system calls are considered native, and the others are supported by an emulation package which translates the additional set of system calls into the corresponding native ones. That approach is usually taken because it is architecturally simple and because most of the expected load will use the native set. There is no particular need to favor one set over the other so far as the internal implementation is concerned, however. In fact, one could even support one system call interface on one machine, another on a different set of machines, and automatically cause a given load module to be executed on the appropriate machine, much in the same manner as done to support heterogeneous cpu instruction sets. (After all, system calls are merely extensions to the basic set of instructions anyway.)

7.3 File System Heterogeneity

Many of the significant problems in operating systems heterogeneity occur in the file systems, for it is there that each system provides an environment which must be globally visible. Thus the nature of the parameters with which one requests functions from the file system is paramount. Unless the file system models are closely compatible, trouble results. Consider an item so simple as the nature of a file name. In Unix, a file name is hierarchical, with each element of the path composed from virtually any ascii character except '/', which is used to separate elements of the path name. While VMS superficially presents a hierarchical path name to applications as well, the first component is a device name (followed by a ':') and the last component is of the form xxxxxx.xxx. Any component is of limited length. As a result, while Unix programs can generate and use any VMS file name, the reverse is not true. If one tries to glue one tree into the other as a subtree, many attempts to create or use files will fail. If the two hierarchies are combined with a new "super root", then those problems still remain, and in addition, any program which used to give what it thought was a complete path name is now missing the initial component (although this is just one more historical problem). Alternately, one can build a general purpose map mechanism that takes a symbolic path name in one naming system and maps it to a name in the other. This approach can give a high level of transparency, but it suffers from the performance degradation implied by the necessity to use bi-directional maps and keep them consistent. This cost need not be great in terms of overall system performance if file system *opens* are relatively rare. However, this map is one of the architectural structures which contributed to NSW's poor performance.[CARL 81] Still other approaches, based on careful use of contexts, are possible when the target operating systems can be altered to provide richer mechanisms.

One also encounters significant differences in the semantics of file system operations and the file types supported. Even if both systems support such types as sequential, multi-indexed, etc., it is often the case that the details of the types differ. For example, one indexed file type might permit variable length keys but no duplicates while the other supports the reverse. This problem is not really as serious as it may seem at first glance. In fact, it is not strictly a transparency issue since it occurs in single machine systems, in the sense that a single site operating system may support a set of file types that are not mutually compatible. In the distributed case we are considering, we merely have a larger set of file types. Of course, it is unpleasant in any environment to have two nearly the same but incompatible facilities.

File system semantic incompatibility is further increased by emerging desires to see such functions as transaction support and multi-object commit supported in the basic system for use across multiple programs, rather than being limited to the database, for example.

Interprocess communication is another important issue. The conventions by which processes dynamically exchange information in a system is another area where lack of agreement causes trouble. If the operating systems across which transparency is desired use ipc mechanisms that cannot be reasonably mapped one to the other, then the situation is not unlike that of multiple similar access types in the file system.

8 Error Modes

It is the conventional wisdom that distributed systems present a significantly richer set of errors to the application and user, and for this reason complete transparency is not possible. In general, that statement is of course true. Individual nodes can fail, taking parts of computations with them which cannot be repeated elsewhere because of special, now lost, resources. However, in our experience, the error situation in distributed systems is often not nearly so complex, for two principal reasons. First, many of the failures in a distributed environment map well into failure modes which already existed in single machine systems. It might be useful to add additional error codes so that the program involved has additional information, but that is hardly a serious change. Even with the additional information, the program's options may be quite limited. For example, if two programs are cooperating, exchanging data, and altering their own states (i.e. their local memory) as a result of data received from the other program, then if one of the pair is destroyed due its host system failing, the other program in most cases will just have to abort.

Second, the additional mechanisms being added to distributed systems to address the admittedly richer error environment are also appropriate for, and are being added to, single machine systems. The best example is the transaction concept. Transactions provide a uniform set of tools for controlling the effect of failures in the distributed environment. Nested transactions have been proposed to limit the effect of failure within a transaction, and now exist in LOCUS [MUEL 81]. Flat transactions have been implemented on single machine systems for some time.

Nevertheless, there are significant differences between errors in the distributed environment and a centralized system, and they can show up in somewhat subtle ways. For example, some errors which are often received immediately in a single machine environment might be received asynchronously, delayed by some amount, in the distributed system. Suppose there is no space on a disk for pages being written by a user process. In a single machine system, an attempt to write the first overflow page generally will cause the application to be immediately signalled. However, in a distributed system, it is rather difficult for the system to notify the user immediately, as there are various delays and buffering in the path from user to remote disk, and from remote machine back to the user. It is quite easy for the user to have written a substantial number of pages before receiving word that none of them can be stored.[1] It is unreasonable to expect that the system will give explicit acknowledgement after each write, as the round trip time implied has unpleasant performance implications.

In this example, while the error is the same, the conditions under which it is reported differ, making the user-specific recovery options more difficult, since a substantial amount of data may have been lost. In the single machine system the user may have been able to keep a few buffers of data, so that in case of error that data could be written elsewhere. This case is an example of the "partial failure" problem mentioned earlier, but here one of the processes involved was the system, while the application continued unaware.

Service outages also make it difficult to maintain completely the illusion of transparency. If communication between sites is lost, then it becomes painfully clear what resources are local and which are remote. There is no perfect solution to this problem; replication of resources helps, but if a replicated storage object is updated on both sides of a broken link, problems may result upon reconnection. If replication is incomplete, then loss of resources will still occur. The handling of partitioned operation when replicated resources are present is a substantial topic in its own right. See [POPE 83, FAIS 83] for an overview. Lastly, programs written for a single machine environment may not be able to usefully deal with the error reports in the network, nor be able to use the added functions like transactions.

9 Local Autonomy and Transparency

It has been well recognized that while a distributed system interconnects a number of sites, those sites may wish to retain a considerable degree of control over the manner in which they operate. The desire for local autonomy is real whether each site is a personal workstation or when the site is a large multiuser system supporting an entire department of users.

One immediately wonders about the relationship between autonomy and transparency. There are multiple issues to be examined:

1. resource control
2. naming

[1] This is especially the case if the system maintains global buffer pools at each site and sends buffers across the net, or initiates writes to disk, only after the pools are filled. While this problem can occur in a single machine system, the delay is worsened in the distributed environment.

3. semantic consistency
4. protection
5. administration

The subject is clearly substantial; here we only briefly touch on the major issues.

9.1 Resource Control

In many cases the machines interconnected in a distributed system are operated by different people in the case of a network of personal computers or by different parts of an organization in the case of larger machines. Each of these different groups typically wish to retain some significant level of control over resource use on their system(s). For example, the personal computer user may well be quite disconcerted if, in the middle of a cpu intensive bitmap update to his screen, some other user in the network dispatched a large task to run on the first user's machine! Transparency makes such an action trivial to take.

The problem is not that transparency is bad; after all, such an action would be possible even through an awkward interface to foreign resources. Rather, resource control tools are needed in any distributed system that connects different administrative domains. Resource controls are also attractive as a way to handle system tuning and load control. If the nature of the controls that are desired is simple, then it may be possible to slightly extend the protection system to accomplish the desired goal. For example, storage space on the local disk could be limited by setting protection controls on the directories whose files are stored on that disk. Control over whether a given user can consume processing resources does require some extension, but in any case, these resource controls are not significantly different in principle from those typically found on single machine systems.

9.2 Naming

The naming problem occurs in the following way. Different users of even the same type of system may configure their directory systems in different ways; on one system the Fortran library is found under one name, while on another system it is elsewhere. If software from these two different systems are to be run in the context of a single, integrated name space, a conflict arises. To solve this problem, one must either force consistent name spaces or use contexts, as described earlier.

9.3 Semantic Consistency

This issue, mentioned early in this paper, is one of the ways that transparency conflicts directly with local autonomy, as we already noted. However, contexts once again help, since the user who wishes to specify a different mapping of name to specific object can do so.

9.4 Protection

The protection problem has several dimensions in a distributed environment. First is the question of what protection controls are actually desired. Are access control lists needed, or will the <owner, group, public> mechanism found in many systems today suffice? We do not discuss this issue here, as it does not seriously impact the distributed system architecture. A second question, however, might. What should a given system trust? That is, machine x receives a message from machine y asking to have file *foo* opened on behalf of user *bar* and a set of pages replaced by updated ones. Should that message be believed? How much does machine x trust machine y? Lindsay [LIND 80c] argues that, at least at the data base level, the most that should be believed is that the message came from machine y. X cannot be sure that the user is really *bar* because the remote software is not trustworthy. Here, we argue that all systems are running the same basic software, and so long as x is sure that machine y is running the standard system, the user authentication can be accepted, and there is no serious problem. Simple, encryption based methods can easily be used to assure that x can tell that the actually loaded software at y is the standard approved version.[1]

9.5 Administration

Large systems require administration and coordination. A distributed, transparent network is no exception. Statistics must be constantly gathered to monitor system operation, detect bottlenecks and failures, etc. Hardware and software must be maintained in a coordinated way. Mechanisms to accomplish these goals are needed, and must be installed and run in each of the participating systems, whether they are wanted or not.

10 Integration of Separate Networks

Despite the attractions of transparency, it is well recognized that when a given system must interface to the outside world (a distributed Unix net connected via SNA to IBM mainframes, say), it will not be feasible to maintain transparency. Hence the more traditional approach of explicit user and application visible protocols will remain. Given that is the case, one could argue that those protocols might as well be used in the local net too. This reduces development, since the transparent mechanism will not be needed. Further, since certain programs will have to operate in a distributed manner over the heterogeneous link, having transparency on one side of that link is no savings. Those programs might as well employ the single global network interface throughout.

[1] Encrypt the system load module and a communication key. Decrypt that load module as part of the booting procedure. Employ the communication key for a digital signature. There are a few additional safeguards actually required to make this procedure safe.

This argument is naive. Much of the transparency mechanism is going to be built anyway by applications; to the extent possible it should be made universally available. In addition, there will certainly be many applications which will not be required to use the heterogeneous link. To force them to do so would be unfortunate indeed. Further, the hypothetical network application mentioned earlier already must use the single machine interface anyway. The transparent extension can largely present the same.

A more interesting case occurs when one has several local networks, each transparent within itself, but where those networks are themselves interconnected by slow links on which a standard protocol such as X.25 or a lower level of SNA is run. How should resources on the distant network look to an application on the local net? We turn to this issue below.

10.1 Long Haul and Low Speed Networks

Geographically dispersed networks predominate by far in today's distributed computing environments. These systems do not provide a high degree of transparency, for three significant reasons. First, many of these networks connect a heterogeneous set of often vendor supplied systems. It is often not practical to modify vendor supported software, and even if it were, differences among the systems make the design of a transparent standard rather difficult. The National Software Works, mentioned earlier, represents one such attempt, that layered a standard interface on top of the system, but below the application. It's incompleteness and poor performance were mentioned earlier.[HOLL 81]

A second important reason for the lack of transparency in existing networks concerns the manner in which they were developed. These systems are generally directly derived from early network architectures in which the most important task was to develop methods by which machines could communicate, rather than to take into account then poorly understood lessons of distributed computing. Third, long haul networks typically represent a scarse resource. They are characterized by either low bandwidth or high delay, and usually by both. Therefore, it is quite appropriate in many peoples' view to make that resource application visible through a different interface than the one through which local resources are seen.

The first issue, heterogeneity, undoubtedly will always be present; in general a high level of transparency in that case is probably not feasible. However, when most of the computers of interest involve the same operating system interface, often the case in local networks or within a given organization, the heterogeneity issue is far less prevalent. The historical reason for the nature of cross system interfaces, while interesting, should not constrain new system architectures. Therefore, we are often left with the nature of the underlying transmission medium as the significant issue to be considered when evaluating the possibility of providing a high level of transparency in long haul environments.

There is no doubt that the media are different; the long haul environment presents far higher delay, much lower bandwidth, and significantly greater error rates than the local net. These differences make absolutely necessary a significant low level protocol which is responsible for error handling (using acknowledgements, retransmission, sequence numbers, etc.), flow control, resource management, connection establishment, name mapping, data conversion, out of band signalling, and the like. These facilities are typically implemented in a multilayered protocol, meeting one of the several international or domestic standards.

Further, existing implementations provide an explicit interface to the network, permitting substantial control over its use. This characteristic is highly desirable, as the net is often a scarse resource, to be managed rather carefully.

However, it is our view that these issues do not conflict with the goal of transparency at all. First, just because a substantial set of protocol modules are needed to manage a complex device (the network) is not a reason to make the interface to the resources accessed through that device different from, and more awkward than, local resources. Similarly one gives a simple open/close, read/write interface to local disks in conventional systems, rather than requiring one to write the complex channel program which ultimately does this work.

The desire for explicit resource control is real, however. Some mechanism is needed to accomplish that goal. We argue that the optimization language described earlier provides a suitable framework, at least from the individual user's point of view. Extensions to that language, to permit a system administrator to set more global resource management policies, undoubtedly will be needed in some environments.

The effect of this transparent approach to long haul networks yields a picture composed of a number of local machine clusters, each cluster internally connected by a local area network, with the clusters interconnected by long haul media. However, this view generates a number of internal architecture problems which must be solved. They include:

a) the unsuitability of such local net based algorithms as page faulting across the local network, as LOCUS does;

b) the flow control problems present in networks where fast and slow links are connected in series (long queues in front of the slow link back up into the fast links, blocking their use as well as associated resources such as buffers);

c) the need for considerable control over location of resources, for optimization reasons, beyond the often simple ones that can be tolerated in a local area environment.

However, these issues are relatively minor compared to the improvement which results from transparency.

11 Conclusions

It is both obvious and easy to say that network transparency in distributed systems, like virtual memory in operating systems or data independence in data bases, is highly desirable. However, the task of achieving a high degree of transparency is far more subtle than it may have appeared at first glance. Many issues raise themselves, as we have tried to indicate in this paper. In addition, there are some inherent limitations which preclude full transparency in numbers of cases. Nevertheless, the benefits of transparency are substantial, and the actual costs (beyond considerable care in design) are relatively modest by comparison.

12 Bibliography

BIRR 82 Birrell, A. D., Levin, R., Needham, R. M., Schroeder, M. D., *Grapevine: An Exercise in Distributed Computing*, CACM, Vol. 25, No. 4, April 1982, pp. 260-274.

CARL 81 Carlson, W., private communication 1981.

ENGL 83 English, Robert M., and Popek, Gerald J., *Dynamic Reconfiguration of a Distributed Operating System*, Submitted to SOSP '83.

FAIS 81 Faissol, S., *Availability and Reliability Issues in Distributed Databases*, Ph. D. Dissertation, Computer Science Department, University of California, Los Angeles, 1981.

GOLD 83 Goldberg, A., and G. Popek, *Measurement of the Distributed Operating System LOCUS*, Submitted to SOSP '83.

GRAY 78 Gray, J. N., *Notes on Data Base Operating Systems*, Operating Systems, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, 393-481.

HOLL 81 Holler E., *The National Software Works (NSW)*, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 421-442.

JONE 82 Jones, M.B., R. F. Rashid and M. Thompson, *Sesame: The Spice File System*, Draft from CMU, Sept. 82.

LAMP 81a Lampson B.W., *Atomic Transactions*, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 246-264.

LAMP 81b Lampson B.W., *Ethernet, Pub and Violet*, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 446-484.

LAMP 82 Lampson, B.W. and H.E. Sturgis, *Crash Recovery in a Distributed Data Storage System*, CACM (to appear)

LEAC 82 Leach, P.J., B.L. Stumpf, J.A. Hamilton, and P.H. Levine, *UIDs as Internal Names in a Distributed File System*, ACM Sigact-Sigops Symposium on Principles of Distributed Computing, Ottawa, Canada, August 18-20, 1982.

LIND 79 Lindsay, B. G. et. al., *Notes on Distributed Databases*, IBM Research Report RJ2571(33471), IBM Research Laboratory, San Jose, CA, July 14, 1979, 44-50.

LIND 80a Lindsay, B.G., *Object Naming and Catalog Management for a Distributed Database Manager*, IBM Research Report RJ2914(36689), IBM Research Laboratory, San Jose, CA, August 29, 1980.

LIND 80b Lindsay, B.G., *Site Autonomy Issues in R*: A Distributed Database Management System*, IBM Research Report RJ2927(36822), IBM Research Laboratory, San Jose, CA, September 15, 1980.

LIND 80c Lindsay, B.G., *R** Notes from the IEEE Workshop on Fundamental Issues in Distributed Systems, Pala Mesa, California, Dec. 15-17, 1980.

LISK 81 Liskov, B. and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust, Distributed Programs*, Computation Structures Group Memo #210, MIT, November 1981.

MEUL 83 Meuller E., J. Moore and G. Popek, *A Nested Transaction System for LOCUS*, Submitted to SOSP '83.

NELS 81 Nelson, B.J., *Remote Procedure Call*, Ph.D. Dissertation, Report CMU-CS-81-119, Carnegie-Mellon University, Pittsburgh, 1981.

PARK 80 Parker, D. Stott, Popek, Gerald J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C., *Detection of Mutual Inconsistency in Distributed Systems*, accepted for publication in IEEE Transactions of Software Engineering, to appear May 1983.

POPE 81 Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G., *LOCUS: A Network Transparent, High Reliability Distributed System*, Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

POPE 83 Popek G.J., G. Thiel and C.S. Kline, *Recovery of Replicated Storage in Distributed Systems*, Submitted to SOSP '83.

RASH 81 Rashid, R.F., and Robertson, G.G., *Accent: A Communication Oriented Network Operating System Kernel*, Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

REED 78 Reed, D. P., *Naming and Synchronization in a Decentralized Computer System*, Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, M.I.T., 1978.

REED 80 Reed, D.P, and Svobodova L, *SWALLOW: A Distributed Data Storage System for a Local Network*, Proc. of the International Workshop on Local Networks, Zurich, Switzerland, August 1980.

RITC 78 Ritchie, D. and Thompson, K., *The UNIX Timesharing System*, Bell System Technical Journal, vol. 57, no. 6, part 2 (July - August 1978), 1905-1930.

SALT 78 Saltzer J.H., *Naming and Binding of Objects*, Operating Systems, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, 99-208.

SPEC 81 Spector, A. Z., *Performing Remote Operations Efficiently on a Local Computer Network*, CACM, Vol. 25, No. 4, April 1982.

STON 76 Stonebraker, M., and E. Neuhold, *A Distributed Database Version of INGRES*, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Memo No. ERL-M612, September 11, 1976.

THOM 78 Thomas, R.F., *A Solution to the Concurrency Control Problem for Multiple Copy Data Bases*, Proc. Spring COMPCON, Feb 28-Mar 3, 1978.

WALK 83a Walker, B.J., *Issues of Network Transparency and File Replication in Distributed Systems: LOCUS*, Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, 1982.

WALK 83b Walker, B.J., and G.J. Popek, *The LOCUS Distributed File System*, Submitted to SOSP '83.

WATS 81 Watson R.W., *Identifiers (Naming) in Distributed Systems*, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 191-210.

LOCUS also provides a number of high reliability facilities, including flexible and automatic replication of storage at a file level, a full implementation of nested transactions[MEUL 83], and a substantially more robust data storage facility than conventional Unix systems.

The heart of the LOCUS architecture is its distributed file system. In this paper, that architecture is described. The essential elements of the file system are its distributed naming catalog, the accompanying synchronization facilities, integrated replicated storage support, and the mechanisms to allow partitioned operation. First a brief overview of the distributed file system is presented. That is followed by a deeper discussion of the system data structures and the procedures used for normal operation. Exceptional conditions and particular problem areas are discussed, and the paper ends with a number of conclusions about the value of integrated distributed file systems and their difficulties of implementation.

2 File system overview

The LOCUS file system presents a single tree structured naming hierarchy to users and applications. It is functionally a superset of the Unix tree structured naming system. There are three major areas of extension. First, the single tree structure in LOCUS covers all objects in the file system on all machines. As noted in the companion paper on transparency,[POPE 83a] LOCUS names are fully transparent; it is not possible from the name of a resource to discern its location in the network. The critical importance of such properties are discussed in that paper. The second direction of extension concerns replication. Files in LOCUS can be replicated to varying degrees, and it is the LOCUS system's responsibility to keep all copies up to date, assure that access requests are served by the most recent available version, and support partitioned operation. Third, the current implementation of LOCUS provides a rich, although still file grained, synchronization policy which is a generalization of 'multiple reader, single writer'.

A substantial amount of the LOCUS file system design, as well as implementation, has been devoted to appropriate forms of error and failure management. These issues will be discussed throughout this paper. Further, high performance has always been a critical goal. In our view, solutions to all the other problems being addressed are really not solutions at all unless their performance is suitable. In LOCUS, when resources are local, access is no more expensive than on a conventional Unix system. When resources are remote, access cost is higher, but dramatically better than traditional layered file transfer and remote terminal protocols permit. Measured performance results are presented in [GOLD 83].

The next sections discuss the static representation of a distributed LOCUS file system. Following sections show how that structure is used to provide transparent resource access.

The LOCUS Distributed File System[1]

Bruce J. Walker and Gerald J. Popek

University of California at Los Angeles

Abstract

LOCUS is a distributed operating system which supports transparent access to data through a network wide file system, permits automatic replication of storage, supports transparent distributed process execution, supplies a number of high reliability functions such as nested transactions, and is upward compatible with Unix. Partitioned operation of subnets and their dynamic merge is also supported.

This paper concentrates on the architecture of the distributed file system component of LOCUS, including the automatic replication of storage. The dynamic behavior of the system, as well as its supporting data structures are outlined, both for normal behavior as well as for recovery and partitioned operation. Various architectural choices and tradeoffs are discussed.

1 Introduction

LOCUS is a Unix compatible, distributed operating system in operational use at UCLA on a set of Vax computers connected by a standard Ethernet. The machine configuration in January 1983 consisted of 17 Vax-11/750s.[2] The system supports a very high degree of *network transparency*, i.e it makes the network of machines appear to users and programs as a single computer; machine boundaries are completely hidden during normal operation. Both files and programs can be moved dynamically with no effect on naming or correct operation. Remote resources are accessed in the same manner as local ones. Processes can be created locally and remotely in the same manner, and process interaction is the same, independent of location. Many of these functions operate transparently even across heterogeneous cpus.

[1] This research was supported by the Advanced Research Projects Agency under research contract DSS-MDA-903-82-C-0189.

[2] Initial work was done on DEC PDP-11/45's using both 1 and 10 megabit ring networks and on VAX 750's using 10 megabit rings.

3 Static Picture of the Filesystem

In order to understand the LOCUS file system, it is helpful to examine the data structures that represent a functioning system. Below, we do so, first discussing the permanently stored structures, and then the dynamically maintained information in volatile memory.

The user and application program view of object names in LOCUS is analogous to a single, centralized Unix environment; virtually all objects appear with globally unique names in a single, uniform, hierarchical name space. Each object is known by its path name in a tree[1], with each element of the path being a character string. There is only one logical root for the tree in the entire network. Figure 1 gives a very small sample of a pathname tree. Note that the dotted lines name the file /user/walker/mailbox.

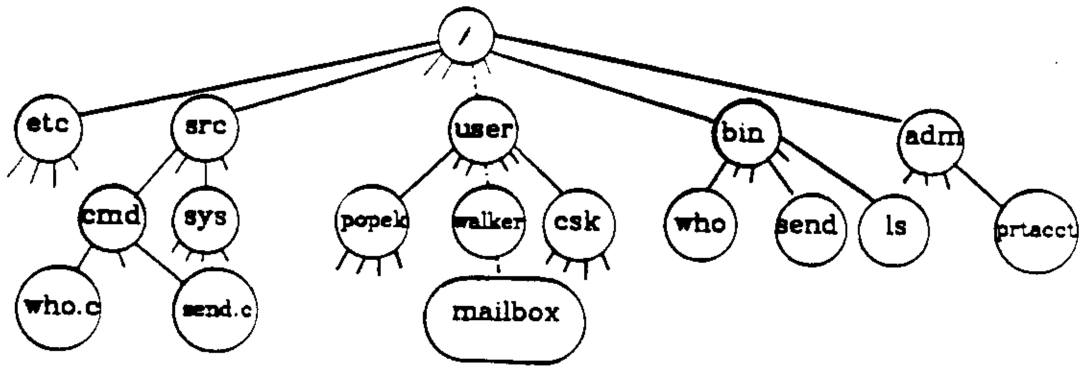


Figure 1: Hierarchical Naming

To a first approximation, the pathname tree is made up of a collection of file groups, as in a conventional Unix environment[2]. Each group is a wholly self contained subtree of the naming hierarchy, including storage for all files and directories contained in the subtree. Connections among subtrees compose a 'super' tree, where the nodes are file groups.

A file group is implemented as a section of mass storage, composed primarily of a small set of file descriptors (called inodes) which serve as a low level internal "directory", and a large number of standard size data blocks. A file is composed of an inode and an associated ordered collection of data blocks. These are used both for leaf or regular file data, and intermediate node or directory file data. One of the directories is treated as the root of the subtree for this file group. Figure 2 sketchs the contents of a filegroup. So far, this portrait is no different than normal Unix. Gluing together a collection of file groups to construct the uniform naming tree is done via the *mount*

[1] There are a few exception to the tree structure, provided by Unix style links.

[2] The term file group in this paper corresponds directly to the Unix term filesystem.

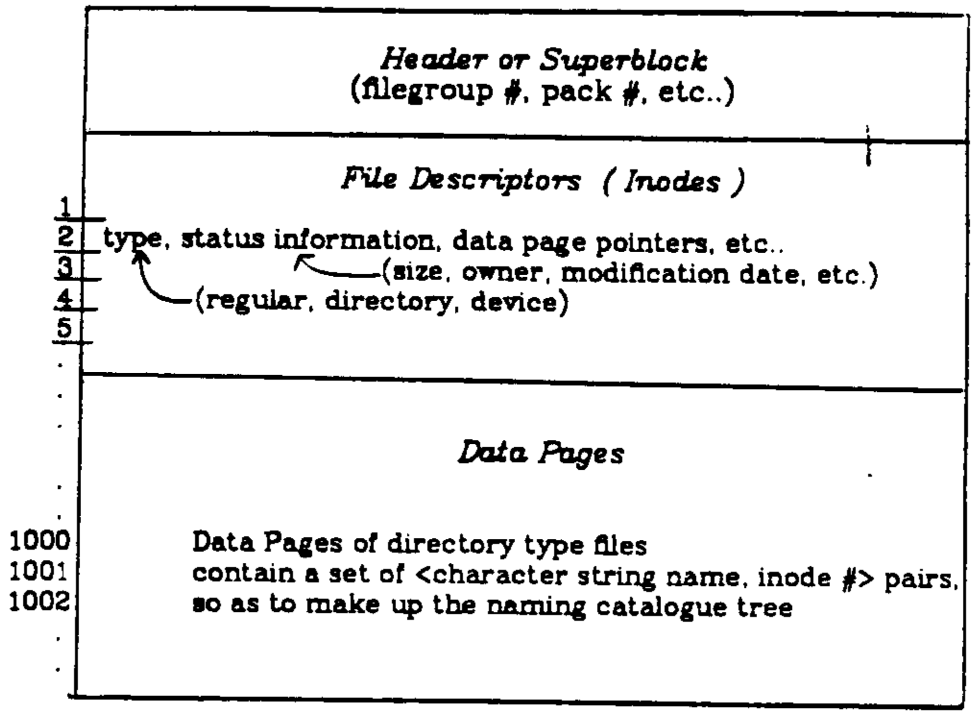

Header or Superblock
(filegroup #, pack #, etc..)
File Descriptors (Inodes)
1
2
3
4
5
type, status information, data page pointers, etc..
(size, owner, modification date, etc.)
(regular, directory, device)
Data Pages
1000
1001
1002
Data Pages of directory type files
contain a set of <character string name, inode #> pairs,
so as to make up the naming catalogue tree

Figure 2: File Group Structure

mechanism. However, the structure of file groups and their inter-relationships is actually more complex in LOCUS than Unix. Complexity is due partly because of the presence of mechanisms for remote operation, but a significant amount of the additional mechanism and data structuring is also present to support replication, so we turn to this subject now.

3.1 Replication

Replication of storage in a distributed file system serves multiple purposes. First, from the users' point of view, multiple copies of data resources provide the opportunity for substantially increased availability.

From the system point of view, some form of replication is more than convenient; it is absolutely essential for system data structures, both for availability and performance. Consider a file directory. A hierarchical name space in a distributed environment implies that some directories will have file name entries in them that refer to files on differing machines. There is powerful motivation for storing a copy of all the directory entries in the path, from the naming root to a given file, local to the site where the file itself is stored, or at least "nearby". Availability is one clear reason. If a directory in the naming path to a file is not accessible because of network partition or site failure, then that file *cannot be accessed*, even though it may be stored locally.

Second, directories in general experience a high level of shared read access compared to update. This characteristic is precisely the one for which a high degree of replicated storage will improve system performance.

3.1.1 Architectural Issues

Support for replicated storage presents a host of problems. One needs highly efficient but themselves replicated mapping tables to find copies of the replicated objects, and those mapping tables must be kept consistent and up to date. Suitable action in the face of failures, to maintain consistent copies of the replicated object as well as the catalog mapping tables, is necessary. It is rather difficult to develop a simple solution to these problems while at the same time addressing the further problems imposed by goals of high availability and performance.

LOCUS replication is designed to:

a. permit multiple copies at different sites, not just on different media on the same machine, although multiple copies at a given site is possible;

b. allow a flexible degree of replication. A given file can be stored on just a single site, at several sites or at many sites, with the possibility of increasing or decreasing the number of copies after initial creation and reasonable freedom to designate where the copies should reside. The system is responsible for keeping all copies consistent and insuring that access requests are satisfied by the most recent copy.

c. allow the user and application program to be as unaware as they wish of the replication system. In other words one can either let system defaults control the degree of replication and file location or one can interact and stategically place file copies.

d. treat all copies of a file as equal. This is not a primary site/backup strategy. Thus as long as one copy is available, operations needing access to the file can continue.

e. support high performance and smooth operation in the face of failures. For example, a running application with a given file open should not be disturbed when another copy of that file reappears in the network.

With these goals in mind, let us consider the static structure that allows replication.

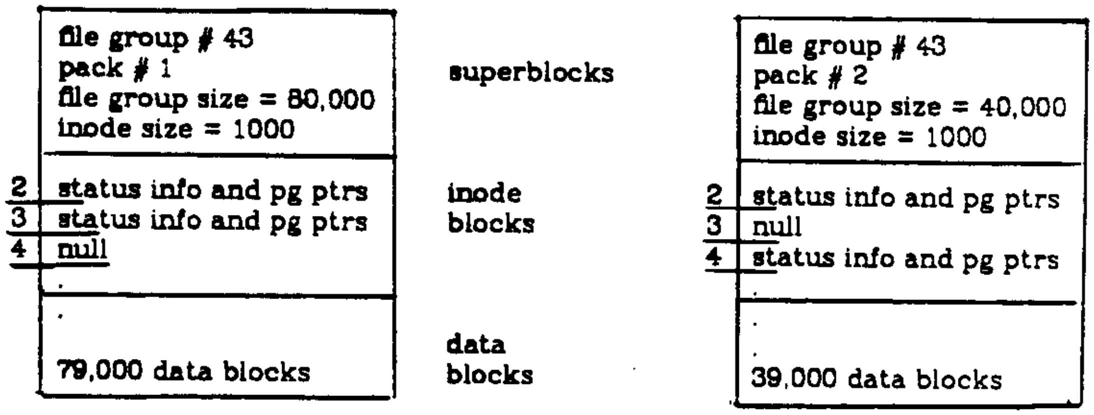
File replication is made possible in LOCUS by having multiple physical containers for a logical file group. Any given logical file group may have a number of corresponding physical containers residing at various sites around the network. A given file belonging to logical file group X may be stored at any subset of the sites where there exists a physical containers corresponding to X. Thus the entire logical file group is not replicated by each physical container as in a "hot shadow" type environment. Instead, to permit substantially increased flexibility, any physical container is incomplete; it stores only a subset of the files in the subtree to which it corresponds. The situation is outlined in figure 3.

From the implementation point of view, pieces of mass storage on different sites are named and maintained as containers for a given file group. Each container is assigned a unique *pack number* and this number, along with the associated logical file group number, is kept in the physical container header (Unix superblock).

To simplify access and provide a basis for low level communication about files, the various copies of a file are assigned the same file descriptor or inode number within the logical file group. Thus a file's globally unique low-level name is:

<logical file group number, file descriptor (inode) number>

and it is this name which most of the operating system uses. Figure 3 gives an introductory sketch of replicated file groups and replicated files.



Note:

- inode #2 is replicated with the same status information on both packs
- inodes #3 and #4 are not replicated

Figure 3: Containers for 2 copies of a File Group

In the example, there are two physical containers corresponding to the logical file group. Note that:

a. the file group number is the same in both copies;

b. the pack numbers are different;

c. files replicated on the two packs have the same file status information in the corresponding inode slot;

d. any file can be stored on either pack or on both;

e. page pointers are not global, but instead maintained local to each pack. References over the network always use logical page numbers and not physical block numbers.

f. the number of inodes on each physical container of the file group is the same so that any file can be replicated on any physical container of the file group[1];

g. if a copy of a file is not stored on a given pack, the status information in that inode slot will so indicate.
h. the inode information stored with each copy indicates not only where other copies are stored but also contains enough history to determine which versions of a file dominate or conflict with other versions. A complete discussion of the ***version vector*** scheme is given in [PARK 80].

The advantages of this replication architecture include:

a. globally unique low level names are implemented, so that high to low level name translation need happen only once in a file access. Inter-machine network traffic uses this terse, efficient name;
b. physical containers can be different sizes;
c. all copies of a file are equal. Therefore if any copy is accessible, normal operation proceeds. Further, from the implementation viewpoint, only one mechanism is needed, instead of one for a primary copy and another for backups, or one for normal operation and another during partitioned mode.[2]
d. a given file need not be replicated, or may have as many physical copies as there are containers for the file group. The decision can be made on a per file basis and the decision can change over time.

Potential disadvantages include:

a. a copy of a file can reside only at those sites which host a pack for the file group;
b. garbage collection of inode slots and sometimes even data pages can be complicated.

LOCUS treats directories as a special type of file. A general replication mechanism was built for files, which normally operates in the same manner, independent of the file type. Additional treatment for handling recovery of directories is provided to automatically merge copies of those directories which were independently updated while a LOCUS system was partitioned. More information can be found in [ENGL 83].

In this section we have concentrated on the LOCUS structure within a single logical file group. We now move on to a discussion of how the super tree is created and maintained.

[1] In the event that one wishes to have a small physical container of a large file group (ie. just store a few of the files), one may not want to pay the overhead of being able to store all the inodes. In that case one could build an indirection mechanism that maps global inode numbers to the smaller local space.

[2] Of course, permitting update when some of the copies are not available can lead to conflicting updates in certain failure modes. The LOCUS solution to this problem is briefly described in a later section. A more extensive discussion appears in [POPE 83b].

3.2 The File Group Mount Table

Logically mounting a file group attaches one tree (the file group being mounted) as a subtree within an already mounted tree. Figure 4a shows a file group with an empty directory "user";

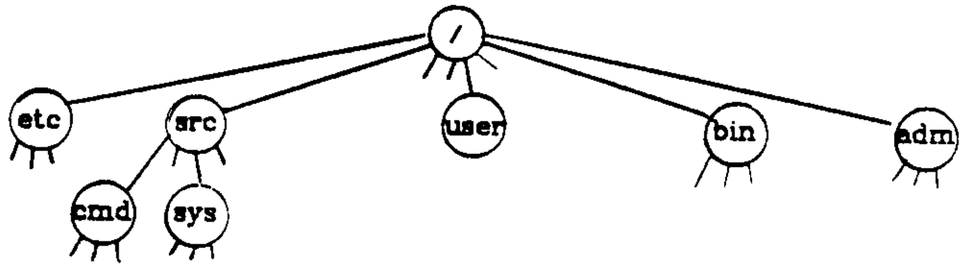


Figure 4a: Pathname Tree Before Mounting File Group 10

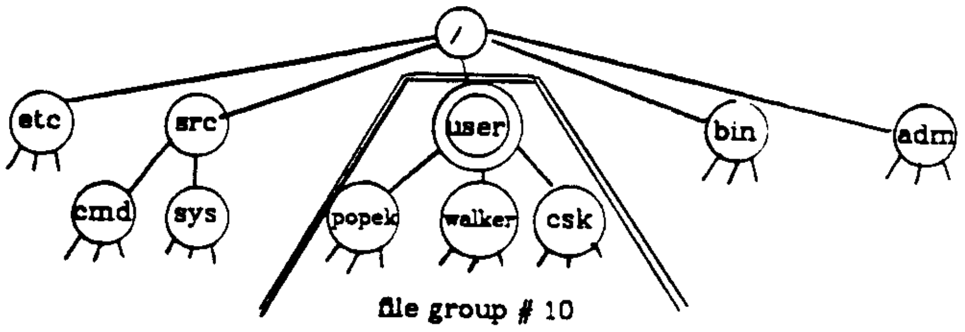


Figure 4b: Pathname Tree After Mounting File Group 10

figure 4b shows the result of mounting a physical container for file group number 10 on top of /user. It is not a constraint that /user be an empty directory but if it were not, its inferior subtree would become hidden and inaccessible after the mount operation.

The mount operation does not change any secondary storage information in either the mounted file group or the "mounted upon" file group. Instead, the glue which allows smooth path traversals up and down the expanded naming tree is kept as operating system state information. In standard, single-site Unix, a table is kept indicating where each file group is mounted. Further, the inode for the directory which is mounted over is kept incore at all times and marked so that requests concerning that directory will be indirected to the initial or root directory of the mounted file group.

In LOCUS this basic mechanism is expanded in several ways. Each machine must have a consistent and complete view of where file groups are mounted; where in the naming hierarchy the subtree was placed, and which sites host packs for the file groups. To maintain the expanded information, the mount table information was split into two tables. The ***logical mount table*** is globally replicated and contains, for each logical file group, the following information:

a. the logical file group number and the inode number of the directory over which the file group is mounted[1].

b. a pair of vectors indicating where (i.e. at what sites in the network) packs of this file group are located.

c. indication of which site is currently responsible for access synchronization within the file group. The role of this site will be described in section 4.2.

In addition, each site which stores a copy of the directory over which a subtree is mounted must keep that directory's inode incore with an indication that it is mounted over, so any access from any site to that directory will be caught, allowing the standard Unix mount indirection to function (now via the logical mount table).

Update of the logical mount table must be done each time a *mount* or *umount* operation is performed or whenever a site joins or leaves the network. There is a protocol implemented within the LOCUS mount and umount system calls which interrogates and informs other sites of mount information. Also, part of the topology change protocol reestablishes and distributes the logical mount table to all sites.

The other part of the mount information is stored in *container tables*. These tables are not replicated. Each site has a container table which holds entries just for those physical containers stored at its site. Local information (eg. a translation from file group number to mass storage location) is kept in this table. Changes are made whenever a physical container of a file group (i.e. a pack) is mounted at that site.

Given the global mount information, unique low level file names (logical file group number, inode number), and container tables, one can now see the basis for a system mechanism to cause users and application programs to view the set of all files on all machines as a single hierarchical tree. We discuss that dynamic behavior now.

4 Accessing the Filesystem

The important functions of the LOCUS distributed file system are creating and removing objects and/or copies of objects, supporting access and modification of those objects, implementing atomic file update, translating path names to physical location, and providing support for remote devices and interprocess communication. Each of these functions is examined below.

[1] The root or base file group is mounted on itself.

4.1 General Philosophy

There were several goals directing the design of the network-wide file access mechanism. The first was that the system call interface should be uniform, independent of file location. In other words, the same system call with the same parameters should be able to access a file whether the file is stored locally or not. Achieving this goal of transparency would allow programs to move from machine to machine and allow data to be relocated.

Good performance was key - both local and remote. Ideally, operations working on data stored locally should not be delayed by any interactions with other machines. Although this objective can often be met, it is sometimes in conflict with the objective of replicated copy consistency. The amount of message traffic was to be minimized, not because of a desire to reduce network load, but because of the software cost of sending and receiving messages. A general message based design was rejected because of the beliefs that: a) the intra-node cost of sending a message could not be made nearly as inexpensive as a corresponding procedure call, b) the local case would dominate system behavior, and c) consequently the overall system performance would be degraded in a non-trivial way if a general message based design were implemented.

Because of the assumption that the underlying network transport medium provided reasonably high bandwidth and low delay, network-wide page faulting was assumed satisfactory. This assumption has worked out well in practice for several different local networks which have hosted LOCUS.

The resulting flow of control structure is a very special case of remote procedure calls. Operating system procedures are executed at a remote site as part of the service of a local system call. Figure 5 traces, over time, the processing done at the requesting and serving site when one executes a system call requiring foreign service. The primary system calls dealing with the filesystem are *open*, *create*, *read*, *write*, *commit*, *close* and *unlink*. After introducing the three logical sites involved in file access and the file access synchronization aspect of LOCUS, these system calls are considered in the context of the logical tasks of file reading, modifying, creating and deleting.

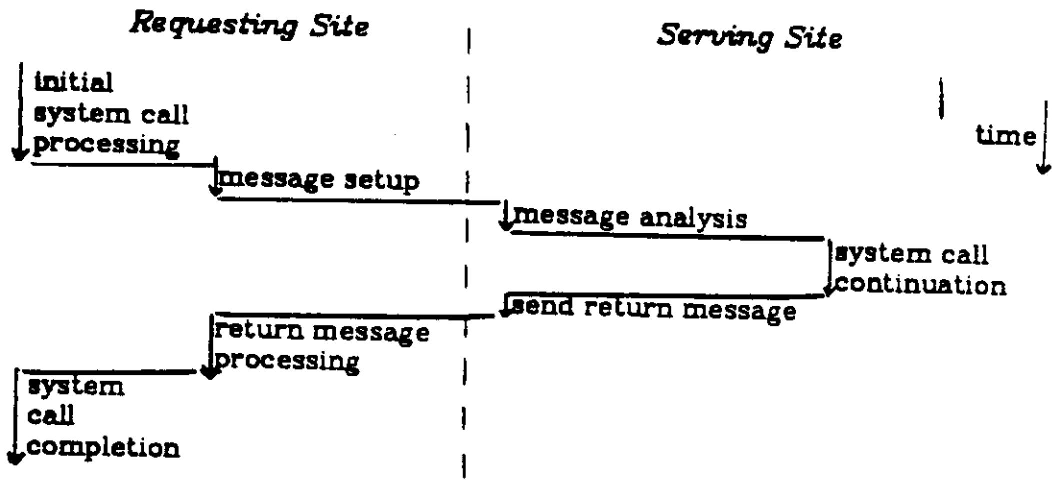


Figure 5: Processing a System Call Requiring Foreign Service

4.2 LOCUS Logical Sites for Filesystem Activities

LOCUS is designed so that every site can be a full function node. As we saw above, however, filesystem operations can involve more than one host. In fact there are three logical functions in a file access and thus 3 logical sites. These are:

a. ***using site***, (US), which issues the request to open a file and to which pages of the file are to be supplied,

b. ***storage site***, (SS), which is the site at which a copy of the requested file is stored, and which has been selected to supply pages of that file to the using site,

c. ***current synchronization site***, (CSS), which enforces a global access synchronization policy for the file's file group and selects SSs for each open request. A given physical site can be the CSS for any number of file groups but there is only one CSS for any given file group in any set of communicating sites (i.e. a partition). The CSS need not store any particular file in the file group but in order for it to make appropriate access decisions it must have knowledge of which sites store the file and what the most current version of the file is.

Since there are three possible independent roles a given site can play (US, CSS, SS), it can therefore operate in one of eight modes. LOCUS handles each combination, optimizing some for performance.

Since all open requests for a file go through the CSS function, it is possible to implement easily a large variety of synchronization policies.

4.3 Synchronization

As soon as one introduces data replication, synchronization is a necessity. In addition, one must resolve how to operate when all the copies are not accessible. There are several alternatives. One could use a primary copy model, where all access must be done to the primary copy and propagated to auxiliary or backup copies. As a variant, secondary copies could be read independent of access to the primary. Another alternative is the majority consensus approach [THOM 78]. Under such a scheme a file can be modified only if the appropriate majority of copies are accessible. Still another approach is to update one copy without any consultation with other copies and then propagate the new version to the other copies. If a conflicting update was done somewhere else, some kind of extraordinary action would be necessary to deal with the problem.[LIND 79] All of these strategies have been proposed or employed, and all have serious operational drawbacks that are discussed in [PARK 80].

In LOCUS, so long as there is a copy of the desired resource available, it can be used. If there are multiple copies present, the most efficient one to access is selected. Other copies are updated in background, but the system remains responsible for supplying a mutually consistent view to the user. Within a set of communicating sites, synchronization facilities and update propagation mechanisms assure consistency of copies, as well as guaranteeing that the latest version of a file is the only one that is visible. The synchronization policy which is currently operational is a generalization of "single writer, multiple reader". Additional options exist partly to permit compatibility with the observed behavior of Unix programs (which typically are run in an environment largely bereft of synchronization control), partly to permit more general behavior in a distributed environment, and partly to support nested transactions. All processes in a process family, or within a given transaction, are treated as a single unit for synchronization purposes.[1]

Since it is felt important to allow modification of a file even when all copies are not currently accessible, LOCUS contains a file reconciliation mechanism as part of the recovery system. Recovering two copies of a file involves checking the *version vectors* of the two copies. If they are the same, the contents of the two copies are the same. If they differ, either one dominates or there is a conflict. A normal propagation is started in the case of domination. A conflict is detected when the file has been modified on two sides of a partition (i.e. on two sites which were not communicating). Conflicting file types like directories and user mailboxes can be reconciled automatically by the system. Other file conflicts are marked as conflicted; the owner is informed and recovery software must be run by the user to reconcile the conflict. More detail on this subject

[1] Synchronization *within* a nested transaction is supported, however. See [MUEL 83].

is given in [POPE 83b].

4.4 Reading Files

To read a file, an application or system supplied program issues the *open* system call with a filename parameter and flags indicating the open is for read. As in standard Unix, pathname searching (or directory interrogation) is done within the operating system open call.[1] After the last directory has been interrogated, the operating system on the requesting site has a <logical file group number, inode number> pair for the target file that is about to be opened. If the inode information is not already in an incore inode structure, a structure is allocated. If the file is stored locally, the local disk inode information is filled in. Otherwise very little information is initially entered.

Next, the CSS is interrogated. If the local site is the CSS, only a procedure call is needed. If not, the CSS is determined by examining the logical mount table, a message is sent to the CSS, the CSS sets up an incore inode for itself, calls the same procedure that would have been called if the US=CSS, packages the response, and sends it back to the US. The CSS is involved for several reasons. One is to enforce synchronization controls. Enough state information is kept incore at the CSS to support those synchronization decisions. For example, if the policy is that only a single site may have a file open for modification at a specific time, the current modification site number is kept incore at the CSS. Another reason for contacting the CSS is to determine a storage site. The CSS stores a copy of the disk inode information whether or not it actually stores the file. Consequently it has a list of packs which store the file. Using that information and mount table information the CSS can select potential storage sites. The potential sites are polled to see if they will act as storage sites.

Besides knowing the packs where the file is stored, the CSS is also responsible for knowing the latest version number. This information is passed to potential storage sites so they can check it against the version they store. If they do not yet store the latest version, they refuse to act as a storage site.

Two obvious optimizations are done. First, in it's message to the CSS, the US includes the version vector of the copy of the file it stores, if it stores the file. If that is the latest version, the CSS selects the US as the SS and just responds appropriately to the US. Another simplying case is when the CSS stores the latest version and the US doesn't. In this case the CSS picks itself as SS (without any message overhead) and returns this information to the US.

The response from the CSS is used to complete the incore inode information at the US. For example, if the US is not the SS then all the disk inode information (eg. file size, ownership, permissions) is obtained from the CSS response. The CSS in turn had obtained that information from the SS. The most general open protocol (all logical

[1] Pathname searching is described in the next section.

functions on different physical sites) is:

US –> CSS	OPEN request
CSS –> SS	request for storage site
SS –> CSS	response to previous message
CSS –> US	response to first message.

After the file is open, the user level process issues read calls. All such requests are serviced via kernel buffers, both in standard Unix and in LOCUS. In the local case data is paged from external storage devices into operating system buffers and then copied from there into the address space of the process. Access to locally stored files is the same in LOCUS as in Unix, including the one page readahead done for files being read sequentially.

Requests for data from remote sites operates similarly. Instead of allocating a buffer and queueing a request for a page from a local disk, however, the operating system at the US allocates a buffer and queues a request to be sent over the network to the SS. The request is simple. It contains the <logical file group, inode number> pair, the logical page number within the file and a guess as to where the incore inode information is stored at the SS.

At the SS, the request is treated, within the operating system, as follows:

a. The incore inode is found using the guess provided;
b. The logical page number is translated into a physical disk block number;
c. A standard low level operating system routine is called to allocate a buffer and get the appropriate page from disk (if it is not already in a buffer);
d. The buffer is renamed and queued on the network i/o queue for transmission back to the US as a response to a read request.

The protocol for a network read is thus:[1]

US –> SS	request for page *x* of file *y*
SS –> US	response to the above request

As in the case of local disk reads, readahead is useful in the case of sequential behavior, both at the SS, as well as across the network.

One of several actions can take place when the *close* system call is invoked on a remotely stored file, depending on how many times the file is concurrently open at this US.

If this is not the last close of the file at this US, only local state information need be updated in most cases. However, if this is the last close of the file, the SS and CSS must be informed so they can deallocate incore inode structures and so the CSS can alter state data which might affect it's next synchronization policy decision. The

[1] There are no other messages involved; no acknowledgements, flow control or any other underlying mechanism. This specialized protocol is an important contributor to LOCUS performance, but it implies the need for careful higher level error handling.

protocol is[1]:

US --> SS	US close
SS --> CSS	SS close
CSS --> SS	response to above
SS --> US	response to first message

Closes of course can happen as a result of error conditions like hosts crashing or network partitions. To properly effect closes at various logical sites, certain state information must be kept in the incore inode. The US of course must know where the SS is (but then it needed that knowledge just to read pages). The CSS must know all the sites currently serving as storage sites so if a certain site crashes, the CSS can determine if a given incore inode slot is thus no longer in use. Analogously, the SS must keep track, for each file, of all the USs that it is currently serving.

4.5 Pathname Searching

In the previous section we outlined the protocol for opening a file given the <logical file group number, inode number> pair. In this section we describe how that pair is found, given a character string name.

All pathnames presented to the operating system start from one of two places, either the root (/) or the current working directory of the process presenting the pathname. In both cases an inode is incore at the US for the directory. To commence the pathname searching, the <logical file group, inode number> of the starting directory is extracted from the appropriate inode and an internal open is done on it. This is the same internal open that was described at the start of the previous section, but with one difference. A directory opened for pathname searching is not open for normal READ but instead for an internal unsychronized read. The distinction is that no global locking is done. If the directory is stored locally and there are no propagations pending to come in, the local directory is searched without informing the CSS. If the directory is not local, the protocol involving the CSS must be used but the locking is such that updates to the directory can occur while interrogation is ongoing. Since no system call does more than just enter, delete, or change an entry within a directory and since each of these actions are atomic, directory interrogation never sees an inconsistent picture.

Having opened the initial directory, protection checks are made and the directory is searched for the first pathname component. Searching of course will require reading pages of the directory, and if the directory is not stored locally these pages are read across the net in the same manner as other file data pages. If a match is found, the inode number of that component is read from the the directory to continue the

[1] The original protocol for close was simply:

US --> SS	US close of file *y*
SS --> US	SS close of file *y*

However, we encountered a race condition under this scheme. The US could attempt to reopen the file before the CSS knew that the file was closed. Thus the responses were added.

pathname search. The initial directory is closed (again internally), and the next component is opened. This strategy is continued up to the last component, which is opened in the manner requested by the original system call.

Some special care is necessary for crossing filesystem boundaries, as discussed earlier, and for creating and deleting files, as discussed later.

4.6 File Modification

Opening an existing file for modification is much the same as opening for read. The synchronization check at the CSS is different and the state information kept at all three logical sites is slightly different.

The act of modifying data takes on two forms. If the modification does not include the entire page, the old page is read from the SS using the read protocol. If the change involves an entire page, a buffer is set up at the US without any reads. In either case, after changes are made, the page is sent to the SS via the write protocol, which is simply[1]:

US —> SS   Write logical page *x* in file *y*

The action to be taken at the SS is described in the next section in the context of the commit mechanism.

The close protocol for modification is similar to the read case. However, at the US all modified pages must be flushed to the SS before the close message is sent. Also, the mechanism at the SS is again tied up with the commit mechanism, to which we now turn.

4.7 File Commit

The important concept of atomically committing changes has been imported from the database world and integrated into LOCUS. All changes to a given file are handled atomically. Such a commit mechanism is useful both for database work and in general, and can be integrated without performance degradation. No changes to a file are permanent until a commit operation is performed. *Commit* and *abort* (undo any changes back to the previous commit point) system calls are provided, and closing a file commits it.

To allow file modifications to act like a transaction, it is necessary to keep both the original and changed data available. There are two well known mechanisms to do so: a) logging and b) shadow pages or intensions lists [LAMP 81a]. LOCUS uses a shadow page mechanism, since the advantage of logs, namely the ability to maintain strict physical relationships among data blocks, is not valuable in a Unix style file system. High performance shadowing is also easier to implement.

[1] There are low level acknowledgements on this message to ensure that it is received. No higher level response is necessary.

The US function never deals with actual disk blocks but rather with logical pages. Thus the entire shadow page mechanism is implemented at the SS and is transparent to the US. At the SS, then, a new physical page is allocated if a change is made to an existing page of a file. This is done without any extra i/o in one of two ways: if an entire page is being changed, the new page is filled in with the new data and written to the storage medium; if the change is not of the entire page, the old page is read, the name of the buffer is changed to the new page, the changed data is entered and this new page is written to the storage medium. Both these cases leave the old information intact. Of course it is necessary to keep track of where the old and new pages are. The disk inode contains the old page numbers. The incore copy of the disk inode starts with the old pages but is updated with new page numbers as shadow pages are allocated. If a given logical page is modified multiple times it is not necessary to allocate different pages. After the first time the page is modified, it is marked as being a shadow page and reused in place for subsequent changes.

The atomic commit operation consists merely of moving the incore inode information to the disk inode. After that point, the file permanently contains the new information. To abort a set of changes rather than commit them, one merely discards the incore information since the old inode and pages are still on disk, and free up page frames on disk containing modified pages. Additional mechanism is also present to support large files that are structured through indirect pages that contain page pointers.

As is evident by the mechanism above, we have chosen to deal with file modification by first committing the change to one copy of a file. Via the centralized synchronization mechanism, changes to two different copies at the same time is blocked, and reading an old copy while another copy is being modified is prevented.[1] As part of the commit operation, the SS sends messages to all the other SS's of that file as well as the CSS. At a minimum, these messages identify the file and contain the new version vector. Additionally, for performance reasons, the message can indicate: a) whether it was just inode information that changed and no data (eg. ownership or permissions) or b) which explicit logical pages were modified. At this point it is the responsibility of these additional SS's to bring their version of the file up to date by propagating in the entire file or just the changes. A queue of propagation requests is kept within the kernel at each site and a kernel process services the queue.

Propagation is done by "pulling" the data rather than "pushing" it. The propagation process which wants to page over changes to a file first opens the file at a site which has the latest version. It then issues standard read messages either for all the pages or just the modified ones. When each page arrives, the buffer that contains it is renamed and sent out to secondary storage, thus avoiding copying data into and out of

[1] Simultaneous read and modification at different sites is allowed for directory files as described earlier.

an application data space, as would be necessary if this propagation mechanism were to run as an application level process. Note also that this propagation-in procedure uses the standard commit mechanism, so if contact is lost with the site containing the newer version, the local site is still left with a coherent, complete copy of the file, albeit still out of date.

Given this commit mechanism, one is always left with either the original file or a completely changed file but never with a partially made change, even in the face of local or foreign site failures. Such was not the case in the standard Unix environment.

4.8 File Creation and Deletion

The system and user interface for file creation and deletion is just the standard Unix interface, to retain upward compatibility and to maintain transparency. However, due to the potential for replicated storage of a new file, the create call needs two additional pieces of information - how many copies to store and where to store them. Adding such information to the create call would change the system interface so instead defaults and per process state information is used, with system calls to modify them.

For each process, an inherited variable has been added to LOCUS to store the default number of copies of files created by that process. A new system call has been added to modify and interrogate this number. Currently the initial replication factor of a file is the minimum of the user settable number-of-copies variable and the replication factor of the parent directory.

Initial storage sites for a file are currently determined by the following algorithm:

a. All such storage sites must be a storage site for the parent directory;
b. The local site is used first if possible;
c. Then follow the site selection for the parent directory, except that sites which are currently inaccessible are chosen last.

This algorithm is localized in the code and may change as experience with replicated files grows.

As with all file modification, the create is done at one storage site and propagated to the other storage sites. If the storage site of the created file is not local, the protocol for the create is very similar to the remote open protocol, the difference being that a placeholder is sent instead of an inode number. The storage site allocates an inode number from a pool which is local to that physical container of the file group. That is, to facilitate inode allocation and allow operation when not all sites are accessible, the entire inode space of a file group is partitioned so that each physical container for the file group has a collection of inode numbers that it can allocate.

File delete uses much of the same mechanism as normal file update. After the file is open for modification, the US marks the inode and does a commit, which ships the inode back to the SS and increments the version vector. As part of the commit mechanism, pages are released and other sites are informed that a new version of the file exists. As those sites discover that the new version is a delete, they also release their pages. When all the storage sites have seen the delete, the inode can be reallocated by the site which has control of that inode.

5 Other Issues

The LOCUS name service implemented by the directory system is also used to support interprocess communication and remote device access, as well as to aid in handling heterogeneous machine types in a transparent manner. We turn to these issues now.

5.1 Site and Machine Dependent Files

While globally unique user visible file naming is very important most of the time, there can be situations where an uttered filename wants to be interpreted specially, based on the context under which it was issued. The machine-type context is a good example. In a LOCUS net containing both DEC PDP-11/45s and DEC VAX 750s, a user would want to type the same command name on either type of machine and get a similar service. However, the load modules of the programs providing that service could not be identical and would thus have to have different globally unique names. To get the proper load modules executed when the user types a command, then, requires using the context of which machine the user is executing on. A discussion of transparency and the context issue is given in [POPE 83a]. Here we outline a mechanism implemented in LOCUS which allows context sensitive files to be named and accessed transparently.

Basically the scheme consists of four parts:

a. Make the globally unique name of the object in question refer to a special kind of directory (hereafter referred to as a *hidden directory)* instead of the object itself.
b. Inside this directory put the different versions of the file, naming them based on the context with which they are associated. For example, have the command */bin/who* be a hidden directory with the file entries *45* and *vax* that are the respective load modules.
c. Keep a per-process inherited context for these hidden directories. If a hidden directory is found during pathname searching (see section 4.4 for pathname searching), it is examined for a match with the process's context rather than the next component of the pathnames passed to the system.
d. Give users and programs an escape mechanism to make hidden directories visible so they can be examined and specific entries manipulated.

5.2 Filesystem Support for Inter-process Communication

Interprocess communication (ipc) is often a controversial subject in a single machine operating system, with many differing opinions. In a distributed environment, the requirements of error handling impose a number of additional requirements that help make design decisions, potentially easing disagreements.

In LOCUS, the initial ipc effort was further simplified by the desire to provide a network-wide ipc facility which is fully compatible with the single machine functions that were already present in Unix. Therefore, in the current LOCUS system release, Unix *named pipes* and *signals* are supported across the network. Their semantics in LOCUS are identical to those seen on a single machine Unix system, even when processes are resident on different machines in LOCUS. Just providing these seemingly simple ipc facilities was non-trivial, however.

Pipes appear in the file system, and may be explicitly opened. They exhibit fifo behavior, with appropriate blocking occurring when the queue is empty or full. However, multiple readers and writers are permitted, and pipe connections are passed as a side effect of creating a child process. If two processes are reading from a pipe, and one reads one character, and the other then reads a character, the second read gets the character which immediately followed the character in the pipe read by the first process. Such behavior is natural when there is shared memory among users of the file; they each access the pipe through a common descriptor. However, since all these processes are potentially located on different machines, an implementation is effectively forced to simulate some characteristics of shared memory in order to ensure correct behavior. The challenge is to do so with high performance in the normal case. Unix application programs typically insert or remove substantial quantities of information from a pipe in a single system call. The LOCUS pipe buffering strategy can involve as many as three sites: the reading site, the storage site, and the writing site, even when there is only a single process at each end of the pipe. Each site has a different view of the buffer and the current queue pointers. In the general case, data flows from the writing site, through the storage site, to the reading site, with optimizations when these site roles are co-located.

5.3 Accessing Remote Devices

In a distributed operating system such as LOCUS, it is desirable that devices, like other resources, be accessible remotely in a transparent manner. In LOCUS, that is the general approach. Device names appear in the single, global naming hierarchy. That is, each device has a globally unique path name, just as other named resources do.

The implementation of transparent remote devices has two important parts. First is the naming and locating part. That is similar to any other type of entry in the catalog system. Pathname search is done until the inode for the device is found. An internal open is done, and a copy of the inode is placed in memory at the storage site (the site where the device is located) as well as at the using site (where the request was made).

The second part of remote device operation is the support for the actual operations that are needed. We distinguish three types of devices: ***buffered block***, ***buffered character*** and ***unbuffered***. A buffered block device is one for which the system is responsible for managing reading and writing of the device in standard sized blocks, and the actual device operation is asynchronous from the caller. Unix block devices, including a number of dma peripherals, fit this category. Support for transparent buffered devices is present in LOCUS in the current implementation. The necessary mechanism is little different from that needed for file support. Remote device access is of course rather valuable. For example, the dump and archive software on LOCUS need not be any different from such software for a single machine Unix system. Compare this situation of "no additional work needed" with the dumping strategies used in less integrated networks. Often, it is the individual users' responsibility because providing a system service is too difficult.

Terminals are the best example of buffered character devices. In the case of a terminal local to the program accessing it, the kernel queues both incoming and outgoing characters, with some processing (eg. echoing) done by the terminal driver portion of the kernel. Remote terminals are not too much different. Data from the terminal is queued only until a remote read message arrives. Then the data is sent to the requesting site and process as a response to the outstanding read. Writes from the remote process are sent to the site of the device for queueing to the device.

There are three major differences between handling remote buffered block and character devices. First, of course is that the number of bytes transferred is variable in the character case. Second, remote read requests in the character case must be handled specially, because it may take an arbitrary time to complete, waiting for user input. Third, it is necessary to provide a means for programs to communicate with remote terminal drivers to indicate the mode in which the terminal is to run. Although code to handle remote buffered character devices is not integrated in the current system, no major problems exist in completing the implementation.

The third class of device is ***unbuffered***. Data is transfered directly from the process's data image to the device. Large data reads or writes can be done in one i/o, so tape drives, for example, are often treated as unbuffered devices. While it is certainly possible to construct a means of accessing remote unbuffered devices, it would require setting up a large buffer space at the device home site and sending across the data one block at a time. One reason for this is that typical local area network hardware cannot handle large messages. By having to buffer at the device site, however, one loses much of the advantage of dealing with an unbuffered device. Instead, if at all possible, one would like in this case to move the process to the data instead of the data to the process. Consequently, LOCUS does not support remote unbuffered device access.

5.4 Node failures

Whenever a node is lost from an operational LOCUS network, cleanup work is necessary. If a CSS for a given file group was lost, another site must take over this role and must gather from all the remaining storage sites of the file group the status of all open files in that file group. In addition, all open inodes involving the lost site(s) must be cleaned up at all remaining sites. This action is the reason why all storage sites keep per file state information indicating which other sites have a given file open. Certain data pages must be discarded, and those processes involved in modification of a remote file which is no longer available must be aborted. Processes reading a remote file whose storage site has been lost may continue if another storage site for that file remains available.[1] There are a number of other actions which are also necessary. See [ENGL 83] for an extensive discussion of these issues.

6 Conclusions

The most obvious conclusion to be drawn from the LOCUS work is that a high performance, network transparent, distributed file system which contains all of the various functions indicated throughout this paper, is feasible to design and implement, even in a small machine environment.

Such concepts as specialized operating system to operating system protocols (as opposed to a layered approach with network server processes), custom remote system calls, lightweight kernel processes, and optimized synchronization protocols are all valuable in achieving this goal.

Replication of storage is valuable, both from the user and the system's point of view. However, much of the work is in recovery and in dealing with the various races and failures that can exist.

Nothing is free. In order to avoid performance degradation when resources are local, the cost has been converted into additional code and substantial care in implementation architecture. LOCUS is approximately a third bigger than Unix and certainly more complex.

[1] That mechanism was not operational in the January 1983 version of LOCUS.

Starting from an operational system such as Unix has been largely beneficial, since many functions did not have to be built. Occasionally, the goal of Unix compatibility made the LOCUS task more difficult. The clearest examples involve synchronization and links.

In summary, however, use of LOCUS (now with tens of thousands of connect hours) indicates the enormous value of a highly transparent, distributed operating system. Since file activity often is the dominant part of o.s. load, it seems clear that the LOCUS architecture, constructed on a distributed file system base, is rather attractive.

7 Bibliography

BARL 81	Bartlett J.F., *A NonStop Kernel*, Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.
BIRR 82	Birrell, A. D., Levin, R., Needham, R. M., Schroeder, M. D., *Grapevine: An Exercise in Distributed Computing*, CACM, Vol. 25, No. 4, April 1982, pp. 260-274.
DION 80	Dion, J., *The Cambridge File Server*, Op. Sys. Rev, 14(4), pp. 26-35, Oct. 1980.
ENGL 83	English, Robert M., and Popek, Gerald J., *Dynamic Reconfiguration of a Distributed Operating System*, Submitted to SOSP '83.
FAIS 81	Faissol, S., *Availability and Reliability Issues in Distributed Databases*, Ph. D. Dissertation, Computer Science Department, University of California, Los Angeles, 1981.
GOLD 83	Goldberg, A., and G. Popek, *Measurement of the Distributed Operating System LOCUS*, Submitted to SOSP '83.
GRAY 78	Gray, J. N., *Notes on Data Base Operating Systems*, Operating Systems, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, 393-481.
HOLL 81a	Holler E., *Multiple Copy Update*, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 284-303.
HOLL 81b	Holler E., *The National Software Works (NSW)*, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 421-442.
JONE 82	Jones, M.B., R. F. Rashid and M. Thompson, *Sesame: The Spice File System*, Draft from CMU, Sept. 82.

LAMP 81a Lampson B.W., ***Atomic Transactions***, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 246-264.

LAMP 81b Lampson B.W., ***Ethernet, Pub and Violet***, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 446-484.

LAMP 82 Lampson, B.W. and H.E. Sturgis, ***Crash Recovery in a Distributed Data Storage System***, CACM (to appear)

LELA 81 LeLann G., ***Synchronization***, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 266-282.

LIND 79 Lindsay, B. G. et. al., ***Notes on Distributed Databases***, IBM Research Report RJ2571(33471), IBM Research Laboratory, San Jose, CA, July 14, 1979, 44-50.

MEUL 83 Meuller E., J. Moore and G. Popek, ***A Nested Transaction System for LOCUS***, Submitted to SOSP '83.

MITC 82 Mitchell J.G. and J. Dion, ***A Comparison of Two Network-Based File Servers***, CACM, Vol. 25, No. 4, April 1982.

NELS 81 Nelson, B.J., ***Remote Procedure Call***, Ph.D. Dissertation, Report CMU-CS-81-119, Carnegie-Mellon University, Pittsburgh, 1981.

PARK 80 Parker, D. Stott, Popek, Gerald J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C., ***Detection of Mutual Inconsistency in Distributed Systems***, accepted for publication in IEEE Transactions of Software Engineering, to appear May 1983.

POPE 81 Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., and Thiel, G., ***LOCUS: A Network Transparent, High Reliability Distributed System***, Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

POPE 83a Popek, Gerald J., and Walker, Bruce J., ***Network Transparency and its Limits in a Distributed Operating System***, Submitted to SOSP '83.

POPE 83b Popek G.J., G. Thiel and C.S. Kline, ***Recovery of Replicated Storage in Distributed Systems***, Submitted to SOSP '83.

RASH 81 Rashid, R.F., and Robertson, G.G., ***Accent: A Communication Oriented Network Operating System Kernel***, Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

REED 78	Reed, D. P., *Naming and Synchronization in a Decentralized Computer System*, Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, M.I.T., 1978.
REED 80	Reed, D.P, and Svobodova L, *SWALLOW: A Distributed Data Storage System for a Local Network*, Proc. of the International Workshop on Local Networks, Zurich, Switzerland, August 1980.
RITC 78	Ritchie, D. and Thompson, K., *The UNIX Timesharing System*, Bell System Technical Journal, vol. 57, no. 6, part 2 (July - August 1978), 1905-1930.
SALT 78	Saltzer J.H., *Naming and Binding of Objects*, Operating Systems, Lecture Notes in Computer Science 60, Springer-Verlag, 1978, 99-208.
SPEC 81	Spector, A. Z., *Performing Remote Operations Efficiently on a Local Computer Network*, CACM, Vol. 25, No. 4, April 1982.
STUR 80	Sturgis, H.E. J.G, Mitchell and J. Israel, *Issues in the Design and Use of a Distributed File System*, Op. Sys. Rev, 14(3), pp. 55-69, July 1980.
SVOB 81	Svobodova, L., *A Reliable Object-Oriented Data Repository For a Distributed Computer*, Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.
THOM 78	Thomas, R.F., *A Solution to the Concurrency Control Problem for Multiple Copy Data Bases*, Proc. Spring COMPCON, Feb 28-Mar 3, 1978.
WALK 83	Walker, B.J., *Issues of Network Transparency and File Replication in Distributed Systems: LOCUS*, Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles, 1982.
WATS 81	Watson R.W., *Identifiers (Naming) in Distributed Systems*, Distributed Systems - Architecture and Implementation, Lecture Notes in Computer Science 105, Springer-Verlag, 1981, 191-210.

Recovery of Replicated Storage in Distributed Systems*

Gerald J. Popek, Greg Thiel and Charles S. Kline
University of California at Los Angeles

ABSTRACT

LOCUS is a distributed operating system which supports replicated storage of user and system files and also permits partitioned operation. Replicated objects may be updated in different partitions. In this paper, these design decisions are discussed, and the actions needed to manage the inconsistencies which can result from such unrestricted replicated operation are described.

1. Introduction

We are concerned with the motivations underlying a desire for replicated storage in a distributed system, the design issues which result, and an architecture plus implementation which demonstrates the feasibility of the solutions proposed. The context is the LOCUS system, in which file replication, network transparency, and support for partitioned operation are provided by the underlying operating system.

We begin this paper with the motivations for replicated storage and the functionality those motivations impose. Several significant problems require solution. Chief among them are synchronization of copies during normal operation, and merging partitions after a failure. Since we will argue that independent update of different copies of replicated data in different partitions is an essential functionality, the merge problem is then significant, since conflicting updates in general will have occurred. In this paper, we concentrate on the merge problem, and outline the solutions which have been adopted for the LOCUS system, as well as for applications in that distributed environment.

1.1. LOCUS Recovery Philosophy

The basic approach in LOCUS is to maintain, within a single partition, strict synchronization among copies of a file so that all uses of that file see the most recent version, even if concurrent activity is taking place on different machines. General update is permitted during partition however; upon merge, conflicts are reliably detected. For those data types which the system understands, automatic reconciliation is done. Otherwise, the problem is reported to a higher level; a database manager for example, who may itself be able to reconcile the inconsistencies. Eventually, if necessary, the user is notified and tools are provided by which he can interactively merge the copies.

An important example where replicated operation is needed, in a distributed system, is the *name service*, the mechanism by which the user sensible names are translated into internal system names and locations for the

* This research has been supported by the Department of Defense under contract DSS-MDA-903-82-C-0189.

associated resource. Those mapping tables must themselves be replicated, for reasons that will be explored below. A significant part of the basic replication mechanism in LOCUS is used by its name service, or directory system, and so we will concentrate on that part of recovery in the remainder of our discussion.

2. Motivation for Replication

Replication of storage in a distributed file system serves multiple purposes. First, from the users' point of view, multiple copies of data resources provide the opportunity for substantially increased availability. This improvement is clearly the case for read access, although the situation is more complex when update is desired, since if some of the copies are not accessible at a given instant, potential inconsistency problems may preclude update, thereby decreasing availability as the level of replication is increased.

The second advantage, from the user viewpoint, concerns performance. If users of the file exist on different machines, and copies are available near those machines, then read access can be substantially faster compared to the necessity to have one of the users always make remote accesses. This difference can be substantial; in a slow network, it is overwhelming, but in a high speed local network it is still signficant.[1]

In a general purpose distributed computing environment, such as LOCUS, some degree of replication is essential in order for the user to be able to work at all. Certain files used to set up the user's environment must be available even when various machines have failed or are inaccessible. The start-up files in Multics, or the various Unix shells, are obvious examples. Mail aliases and routing information are others. Of course, these cases can generally be handled by read-only replication, which in general imposes fewer problems.[2]

From the system point of view, some form of replication is more than convenient; it is absolutely essential for system data structures, both for availability and performance. Consider a file directory. A hierarchical name space in a distributed environment implies that some directories will have entries which refer to files on different machines. There is strong motivation for storing a copy of all the directory entries in the backward path from a file local to the site where the file itself is stored, or at least "nearby". Availability is one clear reason since the entries , in general, will be required to name the file. If a directory entry in the naming path to a file is not accessible because of network partition or site failure, then that file *cannot be accessed*, even though it may be stored locally. LOCUS supports replication at the granularity of the entire directory (as opposed to the entry granularity) to address this issue.

Second, directories in general experience a high level of read access compared to update. As noted earlier, this characteristic is precisely the one for which a high degree of replicated storage will improve system performance. In the case of the file directory hierarchy, this improvement is critical. In fact, the access characteristics in a hierarchical directory system are, fortuitously, even better behaved than just indicated. Typically, the top of the hierarchy exhibits a very high level of lookup, and a correspondingly low rate of update. This pattern occurs because the head of the tree is heavily used by most programs and

1 In the LOCUS system, which is highly optimized for remote access, the cpu overhead of accessing a remote page is twice local access, and the cost of a remote open is significantly more than the case when the entire open can be done locally.

2 The problems which remain are present because few files are strictly read-only; it is just that their update rate is low. When an update is done, some way to make sure that all copies are consistent is needed. If the rate is low enough, manual methods may suffice.

users as the starting point for name resolution. Changes disrupt programs with embedded names, and so are discouraged. The pattern permits (and requires) the root directories to be highly replicated, thus improving availability and performance simultaneusly. By contrast, as one moves down the tree toward the leaves, the degree of shared use of a any given directory tends to diminish, since directories are used to organize the name space into more autonomous subspaces. The desired level of replication for availability purposes tends to decrease as well. Further, the update traffic to directories near the leaves of the naming tree tends to be greater, now increasing the motivation for lessened directory replication to improve performance.

The performance tradeoffs between update/read rates and degree of replication are well known, and we have already discussed them. However, there are other costs as well. For example, concurrency control becomes more expensive. Without replication the storage site can provide concurrency control for the object since it will know about all activity. With replication some more complex algorithm must be supported. In a similar way, with replication, a choice must be made as to which copy of a object will supply service when there is activity on the object. This degree of freedom is not available without replication. If objects move, then, in the no replication case, the mapping mechanism must be more general. With replication a move of an object is equivalent to a add followed by a delete of a object copy.

The remaining performance costs are due to the possibility of failures. One such impact is the maintenance of system objects and internal tables. When a failure occurs, and some copies of objects become unavailable, cleanup must be done and this may involve selecting another copy of the object, restoring a checkpoint, and continuing. A second example concerns the maintenance of system naming structures. These will continue to be modified in the face of partitions and sufficient information must be preserved to produce a consistent structure when both copies are available.

3. Partitions

Partitions clearly are the primary source of difficulty in a replicated environment. Some authors have proposed that the problem can be avoided by having high enough connectivity that failures will not result in partitions. In practice, however, there are numerous ways that effective partitioning occurs. In single local area networks, a single loose cable terminator can place all machines in individual partitions of a single node. Gateways between local nets fail. Long haul connections suffer many error modes. Even when the hardware level is functioning, there are a miriad ways that software levels cause messages not to be communicated; buffer lockups, synchronization errors, etc. Any distributed system architectural strategy which depends for its correct and convenient operation on the collection of these failure modes being exceedingly infrequent is a fragile model, in our judgment.

There are also organizational reasons why effective partitioning can occur. One results from the differential tariffs on communications lines depending on time of day, burst usage, and other considerations. It is often cheaper not to maintain a continual connection between nodes in a network, instead batching updates at night, or when a large amount of traffic has been accumulated. This practice has the same effect as a real partition. Another reason concerns organizational boundaries. Different organizations are often unwilling to permit tight integration of their respective systems. The various airline reservation systems typically operate effectively in a partitioned manner, periodically bringing one another up to date through constrained system interfaces, even though the user view is transparent.

Given partitioning will occur, and assuming replication of data is desired for availability, reliability, and performance, an immediate question is whether a data object, appearing in more than one partition, can be updated during partition. In our judgment, the answer must be yes. There are numerous reasons. First, if it is not possible, then availability goes down, rather than up, as the degree of replication increases. Secondly, the system itself must maintain replicated data, and permit update during partitioned mode. File system directories are the obvious example. Solutions to that problem may well be made available to users at large. Third, in many environments, the probability of conflicting updates is low. Actual intimate sharing is often not the rule, as a result permitting update in all partitions will lead to update in at most one partition anyway, unless the user involved needed to get at an alternate copy because of system failures. In that case he can be kept aware. *To forbid update in all partitions, or all except one, can be a severe constraint*, and *in most cases will have been unnecessary*.

Given the ability to update a replicated object during partition, one must face the problem of mutual consistency of the copies of each data object. Further, the merge procedure must assure that no updates are lost when different copies are merged. Solutions proposed elsewhere, such as primary copy [Alsberg76], majority consensus [Thomas78], and weighted voting [Menasce77] are excluded. They all impose the requirement that update can be done in at most one partition. Even methods such as that used in Grapevine [Birrell82] are not suitable. While Grapevine assures that copies will eventually reach a consistent state, updates can be lost.

It is useful to decompose the replication/merge problem into two cases. In the first, one can assume that multiple copies of a given object may be reconciled independently of any other object. That is, the updates done to the object during partition are viewed as being unrelated and independent of updates (or references) to other objects.

The second case is the one that gives rise to transactions. Here it is recognized that changes to sets of objects are related. Reconciliation of differing versions of an object must be coordinated with other objects and the operations on those objects which occured during partition.

LOCUS takes both points of view. The basic distributed operating system assumes, as the default, that file updates and references are unrelated to other files. The steps which are taken to manage replication under those assumptions are discussed in the next section. In addition, LOCUS provides a full nested transaction facility for those cases where the user wishes to bind a set of events together. Case specific merge strategies have been developed. The recovery and merge implications of these transactions are discussed later.

4. Detection of Conflicting Updates to Files

Suppose file f was replicated at sites S_1 and S_2. Initially assume each copy was identical but after some period sites S_1 and S_2 partitioned. If f is modified at S_1 producing f_1 then when S_1 and S_2 merge the two copies of f will be inconsistent. Are then in conflict? **No.** The copy at S_1 (f_1) should propagate to S_2 and that will produce a consistent state. The copies of the object would be in conflict if during the partition not only was S_1's copy modified to produce f_1 but S_2's copy was modified to produce f_2. At merge a conflict should be detected. As already pointed out the system may be able to resolve the conflict. This is just a simple example. There could be several copies of the object and the history of the modifications and partitions can be complex. Detecting consistency under the general circumstances is non-trivial, but a elegant solution is presented below.

First a notation is presented to represent the partition history of a particular file.

Definition:
A *Partition Graph* G(f) for any file f is a directed acyclic graph (dag) which is labelled as follows: The source node (and the sink node, if it exists) is labelled with the names of all sites in the network having copies of file f, and all other nodes are labelled with a subset of this set of names. Each node can only be labelled with site names appearing on its ancestor nodes in the graph; conversely every site name on a node must appear on exactly one of its descendents. In addition, a node is marked with a "+" if f is modified one or more times within the corresponding partition, and/or a version conflict had to be *reconciled.*

A partition graph for a file represents the history of partitions and modifications of the file (See Figure 1 for an example).

Consistency problems among copies of files are detected by maintaining a vector with each copy of each file. Within every partition (unit of mutual consistency), these vectors keep an update history for the file. As partitions merge, these vectors for the possibly inconsistent files are compared. It turns out that version conflicts are signalled when, and only when, the vectors are "incompatible." We formalize this as follows.

Definition
A *version vector* for a file f is a set of n pairs, where n is the number of sites at which f is stored. The i-th pair $(S_i : v_i)$ gives the index of the latest version of f made at site S_i. In other words, the i-th vector entry counts the number v_i of updates to f made at site S_i. We will use letters A,B,C,... to designate site names, and vectors will be written as {A:9, B:7, C:22, D:3}.

Definition
A set of version vectors are *compatible* when one vector is at least as large as any other vector in every site component for which they each have entries. A set of vectors *conflict* when they are not compatible.

For example, the version vector {A:1, B:2, C:4, D:3} dominates {A:0, B:2, C:2, D:3} so the two are compatible; and {A:1, B:2, C:4, D:3} and {A:1, B:2, C:3, D:4} conflict, but {A:1, B:2, C:4, D:3}, {A:1, B:2, C:3, D:4}, and {A:1, B:2, C:4, D:4} do not conflict, since the third vector dominates the other two. In Figure 2 version vectors are given for f in every partition of Figure 1. The vector {A:2, B:0, C:1, D:0} associated with the node labelled BCD, indicates that f was modified twice at site A, once at site C, and nowhere else. Note in particular that during the {A,B} partition, the file is modified twice at site A. The final merge results in a conflict.

We adopt the following usage of version vectors:

[1] Each time an update to f originates at site S_i, we increment the S_i-th component of f's version vector by one. The vector is committed with the updated file.

[2] File deletion and renaming are treated as file updates. Deletion results in a version of the file of length zero, for example; when all versions of a file are of length zero, information on the file may be removed from the system.

[3] When version conflicts are reconciled within a partition, the S_i-th entry of the version vector for the reconciled file is set to be the maximum of the S_i-th entries of all of its predecessors, and in addition the site initiating the reconciliation increments its entry. This ensures future compatibility with any old versions of the file still remaining on the network.

[4] When copies of a file are subsequently stored at new sites, the version vector is augmented to include the new site information. The definition of compatibility above still applies in this case.

Point [4] states that the vectors are not required to be of fixed length, but may grow as long as the relevant site information is maintained. If a copy of f is added at a site E during some partition, the vector in the partition where the copy was obtained is simply augmented to reflect the existence of the E copy. Thereafter, sites merging with this partition will be required to augment their vectors accordingly. Also, note that the version vectors should be of variable length, so running out of space will not be a problem.

Version vectors serve basically to encode the partial order defined by the partition graph: If one node in the graph "precedes" another, i.e., there is a path from the graph source through the former to the latter, then the version vectors of the two nodes will not conflict.

The version vector of each file copy can therefore be used to detect consistency problems among the copies of an unrelated file in the following way. When a partition merge occurs all the vectors are compared. If two vectors are equal then no action need be taken. If one vector is pairwise greater than or equal to the other then the copy with the larger (dominate) vector should propagate to the smaller (dominated) copy. If neither of the above is true then the copies are in conflict and the system needs to notify the next higher level to resolve the conflict.

For some types of system supported single file structures the system can resolve conflicts mechanically. Directories and mailboxes have relatively simple semantics (add and delete are the major operations) and can be done in this manner. These cases are critical to LOCUS, and will be discussed below.

5. File System Merge

A distributed file system is an important and basic case of replicated storage. The LOCUS file system is a network wide, tree structured directory system, with leaves being data files whose internal structure is unknown to the LOCUS system nucleus. All files, including directories, have a *type* associated with them. The type information is used to by recovery software to take appropriate action. Current types are:

directories
mailboxes (several kinds)
database files
untyped data files

The LOCUS recovery and merge philosophy is hierarchically organized. The basic system is responsible for detecting all conflicts. For those data types that it manages, including internal system data as well as file system directories, automatic merge is done by the system. If the system is not responsible for a given file type, it reflects the problem up to a higher level; to a recovery/merge manager if one exists for the given file type. If there is none, the system notifies the owner(s) of the file that a conflict exists, and permits interactive reconciliation of the differences.

The current LOCUS typing mechanism is very simple and restricted. Data files and associated load modules are marked with one of a set of types. If a program is run that attempts to modify a typed file, and that program is not marked with the same type, then either the attempt fails, or the type of the data file is changed to *untyped*. The only exceptions are certain system utilities like the copy program, which are trusted not to alter the type structure of the data file. Substantial extensions of this mechanism are under way of course. However, this rudimentary facility is enough to permit recovery/merge software to be type specific, and to be assured that the structure of the data objects has been preserved.

6. Transaction Recovery and Merge

We now consider the case where updates and references to objects in the file system were made within a transaction. The generally desired serializable behavior of transactions, even in the face of failures, is well known. Replication raises additional issues, given that goal. The problem is easy to illustrate. Consider partitions P_1 and P_2 and replicated data items I_1 and I_2, each available in both partitions. Transactions T_1 and T_2 are run in partitions P_1 and P_2 respectively. T_1 reads I_1 and writes I_2 as a function of I_1. T_2 reads I_2 and writes I_1 as a function of I_2. Then the partitions merge. Certainly one wishes the resulting value of all data items be the same as it would have been had the transactions been run in a non-partitioned environment. It is not difficult to demonstrate that, in the general case, the only solution is to undo one or both transactions before merge, and then rerun them. Worse, undoing one transaction may require another transaction also be undone. Suppose in the example above, transaction T_3 had run in partition P_1 after T_1 had completed. T_3 read I_2 and wrote I_3 as a function of I_2. Then undoing T_1 in general requires undoing T_3. The obvious domino effect could cause all work which occurred during partition to be undone and then redone; the total computational cost considerably exceeds what would have occurred if operation during partition had merely been halted. Little user inconvenience was avoided by permitting transaction execution during the partition, since no results obtained during partition could be trusted. In fact, even the drastic solution of substantial undo-redo may not be acceptable if some of the transactions have generated irreversible external actions.

Surprisingly however, it is possible, in many cases, to permit operation during partitioned mode and allow the transactions to commit. Upon partition merge, it is feasible to follow merge procedures which are inexpensive, rapid, and that yield a serializable result, without altering any external results.

To aid the reader's intuition, consider an on line banking system. So long as user's accounts do not reach an unacceptably negative balance, it is easy to permit credits and withdrawals to his balance in different partitions. All that is necessary to merge satisfactorily is to have kept the value of the balance at the time that partitioned operation began. Then the merged value of the balance is that initial balance, altered by the two deltas.

This happy state of affairs occurs only if the semantics of the operations which were done during partitioned mode meet a number of axioms which guarantee that merge can be done, in some cases assuming also that certain additional information is kept. In the preceding example, the ahdditional history information was merely the initial account balance, and merge was trivial because the operations during partitioned operation were commutative and compressible.[3]

In fact, it is possible to define different classes of semantics, and for each class define the additional record keeping that is required and give the automatic merge algorithm. Faissol [Faissol81] develops half a dozen such classes, together with the necessary record keeping and merge procedures. For each case, serializability of the result is proven. He further shows that many of the common data base operations fit into the simplest of the classes, with the most efficient of merge algorithms and the most limited of record keeping requirements.

7. Reconciliation of a Distributed, Hierarchical Directory Structure

In this section, we consider how to merge two copies of a directory which has been independently updated in different partitions. Logically, the directory structure is a tree[4] but any directory can be replicated. A directory can be viewed as a set of records, each one containing the character string comprising one element in the path name of a file. Associated with that string is an index that points at a descriptor for a file or directory. In that descriptor is a collection of information about the file. LOCUS generally treats that descriptor as part of the file from the recovery point of view. The significance of this view will become apparent as the reconciliation procedure is outlined.

To develop a merge procedure for any data type, including directories, it is necessary to evaluate the operations which can be applied to that data type. For directories, there are two primitives:

insert (character string path element) and
remove (character string path element).

These operations have rather simple semantics, so that in most cases, automatic merge is easily done. The basic merge algorithm for a given directory, partitioned copies D_1 and D_2 and resulting directory D_r, is as follows.

Let VV_i be the version vector for object i, and let a directory entry in D_1 be denoted $D_1{}^j$. Then:

```
If $VV_{D_1} = VV_{D_2}$ then exit;
If $VV_{D_1} > VV_{D_2}$ then $D_r$ <- A; exit
If $VV_{D_2} > VV_{D_1}$ then $D_r$ <- B; exit
$D_r$ is initially empty
For each $D_1{}^j$ in $D_1$
     if $D_1{}^j$ not deleted, then $D_r$ <- $D_r$ U {$D_1{}^j$}
For each $D_2{}^j$ in $D_2$
     if $D_2{}^j$ not deleted, then $D_r$ <- $D_r$ U {$D_2{}^j$}
```

This basic approach must be augmented, however, to deal with *name conflicts*. If file name F was inserted into D_1 in one partition, and F was also inserted into D_2 in the other partition, special action is required. In LOCUS, the two Fs are considered as different files, and their names are slightly altered to be

3 By compressible we mean that the effect of a series of operations can be replaced by a single one.
4 With the exception of links. See section 7.2.

distinguished before the directory merge algorithm is executed. The owners of the two files are notified by electronic mail that this action has been taken.

In LOCUS, record of deletion is kept in the file descriptor pointed at by a directory entry, instead of in a directory entry.

Further augmentation to the directory merge algorithm must be done because of *links*, but it is best to postpone that subject until file descriptor merge is described.

7.1. Merge of File Descriptors

LOCUS file descriptors, a generalization of Unix inodes, contain a number of items which also must be merged. These include link count, file size, last modification time, last discriptor modification, protection mask, owner identification, group identification, file type, version vector and block pointers. Most of these fields present no problem: they are treated as part of the file. That is, whenever such a field is changed, the file's version vector is incremented, and the intent is that the new version of the file be propagated to all storage sites. This is a reasonable strategy when the propagation algorithm is clever enough to only propagate changes to a file descriptor, rather than the entire file. The LOCUS replicated update protocol operates in that fashion.[5]

There are two information items in the descriptor which require special attention: the delete flag and the link count. There are two cases of interest; when there is only one directory entry in the hierarchy pointing at this descriptor[6] and when there has been more than one. In the first case, if a delete is done, the link count becomes zero, the delete bit is set, the directory entry is removed, and the file's pages are released. The inode is *not* reused until all sites which store the file group (see [Walker83]) containing this file have been notified that the file has been deleted. This step is necessary to assure that an old directory entry doesn't get used to access a new file which has adopted that file descriptor. This problem can be solved in another way, by a change to the directory. Use unique ids, and store them in the directory and descriptor. Most files in LOCUS (and Unix) never have a link count greater than one, and so can be handled in the simple manner just described. However, when the link count is greater than one, significant problems result, and are discussed in the section on links, to which we now turn.

7.2. Links

The naming hierarchy in LOCUS is basically a tree. However, there are several exceptions. Each directory has an entry pointing to itself and an entry pointing to its immediate parent. These cases are handled specially by LOCUS, and cause no particular problems. The remaining exception is significant, however. Any leaf node (i.e. not a directory) in the naming hierarchy can have multiple path names.

The problem which arises concerns deletion. The expected behavior is that when the last directory entry pointing at the file is deleted, only then is the data to be actually deleted. The obvious implementation of such a policy in a single machine is to keep a link-count with the data; when it becomes zero, remove the file. Unfortunately, it is difficult to keep an accurate link count in the face of

5 Doing so in the face of record keeping for commit and the fact that multiple updates may occur before all remote copies have been updated implies that considerable care in this task is needed.

6 Not counting the pointers in the directory which point to itself and its parent. They are handled specially.

network partitions. The problem is that, given replication, the file and various directories may exist in more than one partition. Within each partition, the link count can be independently varied by the addition or deletion of directory entries ("links") pointing at this file. Without keeping a complete history of link changes during partitioned mode, it is very difficult to obtain the correct value at merge time.

There are several straightforward solutions. One of the more radical is to abandon the link count as the means of deciding when to free up file space, using garbage collection instead. Garbage collection has a number of unpleasant effects in a high performance environment, and if the data structures are distributed and not concurrently available, it cannot be done. Alternately, one could restrict the conditions under which links and unlinks can be done, so that record keeping would be easier. One could even abandon this link facility, perhaps replacing it with symbolic links as in Multics [Organick72]. This step would not maintain compatibility with Unix, of course. Unfortunately, none of these obvious methods is fully satisfactory. Garbage collection is totally unworkable, since some of the pointers which must be checked may not even be available in the current partition, and space may be needed before merge. Compatibilty with Unix links is important, since they are heavily used for a small number of files, and so restrictions on use, or the change in semantics implied by symbolic links, represent significant change.

In LOCUS, substantial additional record keeping is being implemented to permit a general solution. Essentially, a history of link activity during partitioned operation is being provided so that deletion and link merge can automatically be done.

In Section 7 a simple algorithm was presented for merging directories when each directory entry is associated with exactly one file. When links are introduced so that multiple entries exist for each file then the algorithm becomes more complex. The major problem is the lack of a delete bit in the directory entry. In UNIX a deleted entry is indicated by a zero value for the descriptor number. But in order to do the merge it is essential to remember the descriptor number since it is the only way to detect name conflicts. Therefore, in order for merge to operate properly deleted entries must maintain the descriptor number and a delete indication. It is also important for the unique part of the entry name to be retained. This is due to the existence of multiple links to the same descriptor in the same directory.

The alternation to the merge algorithm can be decomposed into an exhaustive set of cases and a number of simple observations. The cases and observations are given below, assuming D_1 and D_2 are the partitioned copies of a directory.

The first observation is that the resolution of an entry F is independent of all entries with entry names not equal to Fs.

Now consider all entries from D_1 and D_1 with the same entry name, in turn. Separate them into groups of equal descriptor numbers. If the resolution of more than one group results in an entry being saved then a name conflict has occurred (For example, this can occur by creating two files with the same name in distinct partitions). If only one group produces an entry for the result directory then it is appended. If no groups produce an entry then no entry is kept.

All entries within a group have equal entry names and descriptor values and therefore only the delete bit could vary among the entries. If there is only one entry in the group then that result is the entry if the delete bit is off, otherwise nothing. If there are two entries[7] in the group then there are four possible

7 A unlink-link sequence of an existing entry results in the unsetting of the delete bit in the original entry. Thus there can be at most two entries with the name name and descriptor

combination of their delete bits (off-off, off-on, on-off and on-on). Only the first results in a saved entry, all others give no saved entry.

The outline presented above indicates how directory merge is performed in the face of links. The next section outlines how reconciliation of mailboxes is performed.

8. Reconciliation of Mailboxes

Automatic reconciliation of user mailboxes is important in the LOCUS replication system, since notification of name conflicts in files is done by sending the user electronic mail. It is desirable that, after merge, the user's mailbox is in suitable condition for general use.

Fortunately, mailboxes are even easier to merge than directories. The reason is that the operations which can be done during partitioned operation are the same: insert and delete, but it is easy to arrange for no name conflicts, and there are no link problems. Further, since mailboxes are not a system data structure, and generally are seen only by the small number of mail programs, support for deletion information can be easily installed.

Thus, for each different mail storage format[8] there is a mail merge program that is invoked after the basic file system has been made consistent again. These programs deal with conflicted files detected by the version vector algorithm which the typing system indicates are mail files.

9. Conflicts Among Untyped Data Objects

When the system has no mechanisms to deal with conflicts, it reports the matter to the user. In LOCUS, mail is sent to the owner(s) of a given file that is in conflict, describing the problem. It may merely be that certain descriptive information has been changed. Alternately, the file content may be in conflict. In any case, files with unresolved conflicts are marked so normal attempts to access them fail, although that control may be overridden. A trivial tool is provided by which the user may rename each version of the conflicted file and make each one a normal file again. Then the standard set of application programs can be used to compare and merge the files.

10. Other Name Spaces

The operating system is not the only environment which must provide a distributed name space with replicated objects. The name space of a data base management system is in many respects quite similar. If the underlying system is a distributed operating system (DOS) providing a global and transparent name space and the replication and recovery support described above, then substantial advantage may be obtained by an appropriate mapping of the distributed data base management system's (DDBMS) name space onto the DOS's name space. This section indicates how that mapping can be done.

First, the nature of the DDBMS name space is briefly outlined. It should be global and transparent [Popek83] in order to provide flexibility and ease of use. Two considerations suggest it should be hierarchical in nature. First, in a large distributed system name collisions at the data base level could be frequent (for example, many *employee* data bases). A hierarchical name space would avoid

value at a partition merge.

8 There are two storage formats in LOCUS; one in which multiple messages are stored in a single file, the default, and another where each message is a different file, and messages are grouped by parent directory. This second storage discipline is used by the mail program *mh*.

this problem. Second, if the DDBMS supports partitioned relations[9], a hierarchical relationship among them is natural. These considerations suggest the DDBMS should provide a global, transparent and hierarchical name space. The next section discusses how to map that name space onto similar name space of a distributed operating system.

10.1. DDBMS To DOS Name Mapping

If the DDBMS's name space is directly mapped to the DOS's name space then DOS's name mechanisms can provide most of the name support. For example, the directory recovery mechanism would perform the merge and conflict detection of the DDBMS's name catalogue required after network reconnection. In this way the DDBMS does not rebuild much of the functions of the DOS. Also, with one common mechanism improvemetns are shared by both users. Third, performance enhancements of the DOS name translation mechanisms will have direct impact on the DDBMS.

The alternative is for the DDBMS to build its own name space within a few files of the DOS. If the DDBMS name space is implemented by structure within a few DOS objects, then the DDBMS must duplicate many of the functions provided by the DOS. It must deal with the replication of the appropriate entries, partition merges of the catalogues, etc. On the other hand performance tuning is DDBMS specific. The catalogue could look much like a normal relation. Representation of the hierarchy may be confusing when displayed in this fashion, however.

There are other DDBMS catalogues, such as protection catalogues, which share similar characteristics with the naming facility. They benefit in a similar fashion from use of the underrlying system directory management facilities.

10.2. DOS Requirements

Mapping the DDBMS's name space onto the DOS's name space places several requirements on the DOS's name support mechanism.

First the user needs to be able to associate some variable length data with each directory entry in order to store the catalogue information. The user must be able to change the data (update the catalogue information) and the system must be able to detect conflicting updates to the data. This implies a version vector be associated with the directory *entry*. Lastly, the DDBMS must be able to easily specify a particular entry. This suggests that there should be a directory entry identifier (The equivalent functionality to a tuple identifier, if the catalogue were a real relation). This will permit the DDBMS to have tuple identifiers for all relation and catalogue tuples and only at the access method level need it be concerned about representation, in normal operation. Depending on the granularity of the replication provided by the distributed operating system it may be essential that symbolic links also be provided.

10.3. A Proposed Mapping

Given the DOS functions proposed above the DDBMS name space can be supported in the following manner.

For each node in the DDBMS hierarchy there will be a directory in the DOS file system. The children of the node will be entries in the directory. In

9 Partitioned relations provide, in normal operation, the ability to store portions of relations at sifferent sites, and yet have the data base system make its composition transparent to the user.

addition each entry will contain any additional data describing the child node. All other catalogues will be maintained in a child directory with tuples being directory entries in the catalogues's directory. The catalogue's information will be stored in the user supplied data of the directory entries that comprise the catalogue. In order for the recovery mechanism to perform properly the name of a directory entry must be a unique key for the catalogue.

Symbolic links will be used whenever the replication factors for a node requires the entry be created in another part of the distributed operating system's name space. This should only be necessary if arbitrary replication is not provided at the granularity of a file.

10.4. Recovery Algorithms

In Section 10 it was suggested that directory merge algorithms might be used to merge database catalogues. Here a presentation of a catalogue merge algorithm is given Four basic operations are performed on a data base catalogue. They are:

insert (catalogue entry),
remove (catalogue entry),
readentry (key) and
updateentry (key, domain, newvalue).

These four operations comprise a basic set of the operations that the database may perform since more complex operations can be formed as compositions of the four above. The first two operations add and delete catalogue entries. The third operation permits the DDBMS to examine a catalogue entry and the last operation provides a means for the DDBMS to change a field of an entry.

The first two operations are essentially equivalent to the insert and remove operations of the directory merge algorithm. The only difference is that entries in different catalogues may be related, while in the operating system directory case all entries were assumed to be independent of one another. This is not a problem since all catalogue entries will have a unique key as the entry name used by the directory merge algorithm. Name conflicts will only occur when the user creates two relations with the same name. Conflicts of this sort must be resolved by discarding or renaming one of the two relations. At this point the DDBMS must have sufficient backpointers to properly resolve the catalogue entries for each relation.

The read operation by itself presents no problems, but the update operation, by itself or in combination with read causes difficulty. In general this is the problem being addressed by Faissol's work [Faissol81], but due to the limited number of operations which may exist (only those defined by the DDBMS) and the fact the DDBMS is executing the operation, itself, it may be possible for simplifications to be made. The exact nature of the simplifications will depend on the operations and the importance of having exact data in the catalogue. One possible simplifaction is to consider how critical it may be to have exact data in the catalogue. Consider a catalogue update of the last modification time for a relation. A merge of a catalogue with conflicting times for a particular relation's last modification could be resolved by selecting the most recent time. This simple merge algorithm results from the DDBMS's total understanding of the implications of the field in the catalogue and it's use.

10.5. FORDELE on LOCUS; A Case Study.

FORDELE is a DDBMS being developed at UCLA. The approach used in the development was to extend the DBMS Ingres [Stonebraker76] to be distributed and run it on top of LOCUS instead of UNIX [Ritchie74]. Because of the global, transparent nature of LOCUS, Ingres will operate in a semi-distributed mode without modification. For more details of on FORDELE see [Thiel83].

Most of the current prototype implementation of FORDELE has involved implementation of the the catalogue management work described above. The directory structure of the LOCUS operating system was extended in the manner described in Section 10.2. At the time of the initial implementation the authors did not understand that all the catalogues could be supported by the LOCUS name space so the implementation in progress only implemented the naming catalogue of FORDELE within the LOCUS name space. Ingres has six catalogues; which are: the relation catalogue, the attribute catalogue, the protection catalogue, the integrity catalogue, the tree catalogue and the index catalogue. The relation catalogue is the Ingres catalogue that supports the name space of Ingres and is the one FORDELE mapped onto the LOCUS name space.

The types of operations performed on the relation catalogue are the addition and deletion of tuples when relations are created and destroyed. Also the read and update operations are performed on catalogue entries since the catalogue contains information like tuple counts and purge dates. Neither of these two items need to be exact, it can be argued, since the former is only used for query optimization decisions and the later need only be the most optimizitic of conflicted values. Incorrect query optimization decisions will only result in degraded performance and not incorrect behavior and therefore the tuple count need not be exact. Occasionally an application can run and cause the count to be correctly updated by examining the data base. As examples of operations that must be maintained correctly by the DDBMS are those operations which effect the protection information in the protection catalogue. For example, one field in the catalogue specifies which terminals are permited for a user to use in referencing a particular relation. When this field is updated the DDBMS must guarentee a partition merge does result in the change being correctly propagated.

11. Conclusions

It is our experience that replication is important in a distributed environment, and that a liberal update policy is very valuable. However, developing the appropriate architecture and mechanism is a non-trivial task. It is clearly feasible to do so, as the LOCUS implementation demonstrates.

Our experience also suggests the name service functions which occur at different levels in a system share many common elements, and there is considerable promise in providing a reasonably general name service function which can support a distributed file directory mapping system, a database catalog system, etc.

There are also a number of ways in which the nature of the name service to be supported affect the ease with which a replicated implementation can be constructed, and modest differences in function can have dramatic effect. The clearest case in LOCUS are file links.

Lastly, it is striking that update of a replicated data item in different partitions can be permitted as often, and in as general a set of circumstances, as we have found. This is a pleasant surprise indeed.

Acknowledgements

The authors gratefully acknowledge the entire LOCUS research group, without whose efforts the portion of the system reported here, and therefore this paper, would not exist.

12. References

Alsberg, P. A., Day, J. D., *A Princple for Resilient Sharing of Distributed Resources*, Proceedings of Second International Conference on Software Engineering, October 1976.

Birrell, A. D., Levin, R., Needham, R. M., Schroeder, M. D., *Grapevine: An Exercise in Distributed Computing*, Communications of the ACM, Vol. 25, No. 4, April 1982, pp. 260-274.

Faissol, S., *Availability and Reliability Issues in Distributed Databases*, Ph.d. Dissertation, Computer Science Department, Unversity of California, Los Angeles, August 1981.

Menasce, D. A., Popek, G. J., Muntz, R. R., *A Locking Protocol for Resource Coordination in Distributed Systems*, Technical Report UCLA-ENG-7808, Dept. of Computer Science, UCLA, October 1977.

Organick, E. I., **The Multics System: An Examination of its Structure**, MIT Press, Cambridge, MA, 1972.

Popek, G. J., Walker, B. J., *Transparency in Local Area Networks*, Submitted to SOSP '83.

Ritchie, D., *The Unix Time-Sharing System*, Communications of the ACM, Vol. 17, No. 7, July 1974, pp. 365-375.

Stonerbraker, M., Wong, E., Kreps, P., *The Design and Implementation of Ingres*, ACM Transactions on Database Systems, Vol. 1, No. 3, Sept. 1976, pp. 189-222.

Thiel, G., *Partitioned Operation and Distributed Data Base Management System Catalogues*, Ph.d. Dissertation, Computer Science Department, Unversity of California, Los Angeles, June 1983.

Thomas, R. F., *A Solution to the Concurrency Control Problem for Multiple Copy Data Bases*, Proceedings Spring COMPCON, Feb 28-Mar 3, 1978.

Walker, B. J., Popek, G. J., *The LOCUS Distributed File System*, Submitted to SOSP '83.

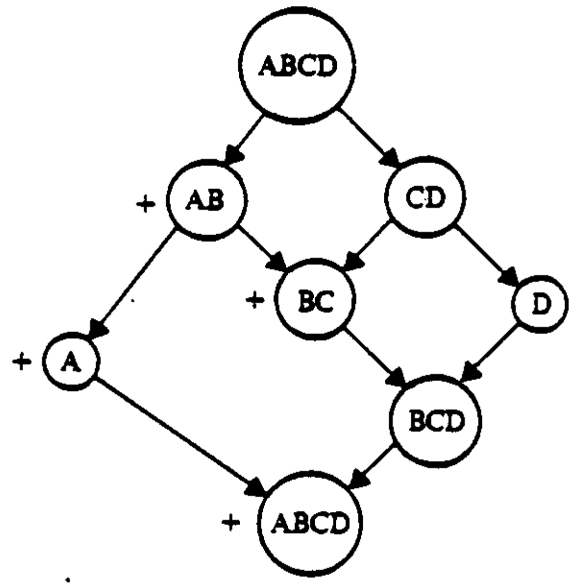


Figure 1. Partition graph G(f) for file stored redundantly at sites A, B, C, D.

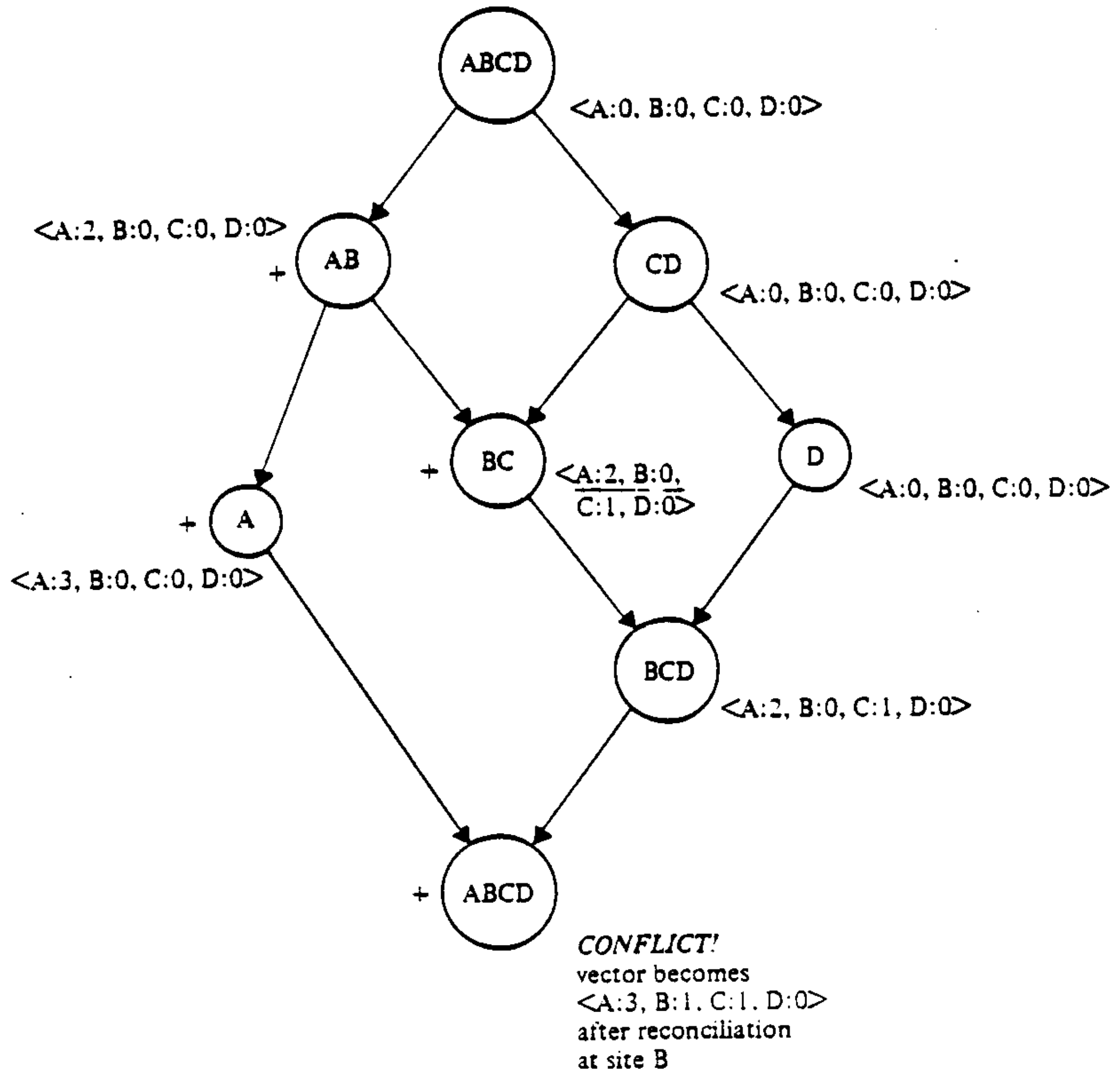


Figure 2. Partition graph G(f) for f with version vectors effective at the end of each partition.

Dynamic Reconfiguration of a Distributed Operating System[1]

Robert M. English and Gerald J. Popek

University of California at Los Angeles

Abstract

LOCUS is a distributed operating system which provides a very high degree of network transparency. In addition, substantial high reliability, availability and recovery mechanisms are provided, including automatic user controlled replication of files, nested transactions, and automatic reconfiguration of the network. Because LOCUS gives the application the illusion of a single machine, there are subsantial relationships among the system data structures in different computers in the network. System protocols also assume connectivity of the transmission medium. For these reasons, the task of dynamically reconfiguring the network as machines leave and arrive, or as the network becomes partitioned, is essential to LOCUS.

In this paper, the reconfiguration architecture of LOCUS is described. The protocols are presented by which a set of machines agrees on the membership in a partition, maximizing that set while still running rapidly, and operating correctly in the face of various continuing failures and race conditions. Then the actions taken to recover internal data structures are briefly outlined.

1 Introduction

The common arguments for distributed operating systems—incremental growth potential, decreased costs—are valid only if the software costs on a distibuted system are comparable to those on a single machine operating system. It does little good to provide incremental growth capabilities in hardware when the software must be modified with each change. A system requiring explicit network handling affects every application program, multiplying the network costs.

LOCUS was designed to be a network transparent, high reliability, high availability distributed operating system. A primary goal was to minimize network related programming costs by providing a consistent user interface independent of transient network topologies. LOCUS provides, for example, file replication for reliability, and allows full operation even in a partitioned environment. File names are independent of the storage location: with portable media, a group of files can even be moved from one site to another in a matter of minutes without the change being noticed by user programs.

Transparency in LOCUS applies not only to the static topology of the network, but to the configuration changes themselves. The system strives to insulate the users from reconfigurations, providing continuing operation with only negligible delay. As in the static case, requiring user programs to deal with such situations shifts the network costs from the operating system to the applications programs.

[1] This research was supported by the Advanced Research Projects Agency under research contract DSS-MDA-903-82-C-0189.

This paper discusses the concept of transparency as it relates to a dynamic network environment, gives several principles that the operating system should follow to provide it, and presents the reconfiguration protocols used in LOCUS. The protocols make use of a high-level synchronization strategy to avoid the message overhead of two-phased commits or high-level ACKs, and are largely independent of the specific architecture of LOCUS. The discussion concludes with an overview of future plans.

2 Other Work

The degree to which the details of an operating system design affect the reconfiguration strategy makes comparisons difficult. Different design goals, philosophies, and assumptions can lead to widely varied definitions of the problem, and even wider variations in the solution. It may, in fact, be unwise to speak of "reconfiguration architectures" except as solutions to specific problems imposed by particular operating systems, and as yardsticks for comparing various system designs.

In the Medusa system at Carnegie-Mellon, for example, individual processors have only fragments of the operating system. Robustness in the face of all possible failure conditions was considered unreasonable: "it was never a goal to respond to all possible failures or changes in environment...Massive failures... are both infrequent and very difficult to overcome [Ousterhout80]." The degree of interaction between nodes makes the full solution impractical. Medusa-type systems could be classified as multi-processor systems: while they do not share primary memory, they depend upon a high degree of interaction and a highly reliable communications bus for correct operation.

At the other end of the spectrum, the Grapevine system developed by Xerox binds nodes together so loosely that reconfiguration never becomes an issue [Birrell82]. No overall network status is maintained; correct operation depends on site-to-site protocols. In Grapevine, and other network environments such as the ARPAnet, correct operation of the nodes depends solely on local status, and while network failure can interrupt network services, it does not affect local service.

We characterize network operating systems by complete replication of local functionality, and separation of local and remote operations; multiprocessors by dependence on communications for most operations. A distributed system such as NSW[Millstein 77] lies somewhere between the two extremes.

LOCUS also falls somewhere between these two extremes: each node has a self-sufficient operating system, but a high degree of interaction takes place to provide file replication, synchronization, and inter-node communication. The system requires explicit reconfiguration strategies to handle failures, but a full solution of the problem is feasible because the effects of failures are limited by the architecture.

3 The LOCUS Operating System[1]

[1] This section reviews those aspects of LOCUS relevant to this paper. Readers familiar with LOCUS can safely skip this section.

The distributed operating system LOCUS runs on a high bandwidth local area network. Derived from Unix,[1] it provides application code compatibility with Unix user programs in an environment of network transparency, enhanced reliability, and high availability. To the user a LOCUS system appears, to the first approximation, like a large, single-site Unix machine. The network is invisible to normal user activity, although additional system calls are available that give network status information.

The LOCUS file system presents a single tree-structured naming hierarchy to users and applications. It is functionally a superset of the Unix tree-structured naming system. There are three major areas of extension. First, the single tree structure in LOCUS covers all objects in the file system on all machines. As noted in the companion paper on transparency,[Popek83] LOCUS names are fully transparent; it is not possible from the name of a resource to discern its location in the network. The critical importance of such properties are discussed in that paper. The second direction of extension concerns replication. Any file in LOCUS can be made replicated, and it is the LOCUS system's responsibility to keep all copies up to date, assure that access requests are served by the most recent available version, and support partitioned operation. Third, the current implementation of LOCUS provides a rich, although still file grained, synchronization policy. That policy, a generalization of 'multiple reader, single writer', permits related processes to interact intimately and also supports full nested transactions.[Mueller83]

Underlying the user hierarchy is a global system hierarchy composed of *file groups*.[2] Like the file tree, each node of the file group tree may be replicated. File groups may be replicated, but only one copy of a file group may be stored at a particular site. The replication factor of a file is limited to that of its file group. File groups are joined by the *mount* system call, which "mounts" a file group on a file in another file group, forming a file group hierarchy in which the file groups play roles analogous to that of the files in the naming tree. The root of the hierarchy is fixed when the system boots. The tree structure is maintained by a local structure called the *mount* table, which records where a file group is mounted in the hierarchy, where copies of the file group are located, and, if the file group is stored locally, where the file group is physically stored. A mount table reflecting the new hierarchy can always be rebuilt from the information available in local partitions after any configuration change.

The physical storage of a file group pack is split into a data and a directory section. Each file has its own data descriptor called an inode. The index of an file's inode in the inode space is unique within a file group, and points to the file in the directory. The system uses the file group number, inode number pair as a unique global internal name for the file.

In keeping with the philosophy of network transparency, the system allows full operation regardless of the current system configuration. In particular, the system places no barriers to concurrent modification to files in separate partitions. In the Unix environment few resources are concurrently shared, and the increase in availability gained by allowing partitioned operation outweighs the costs of detecting and

[1] Unix is a trademark of Bell Laboritories.

A LOCUS file group is nearly the same as a Unix file system.

resolving any resultant conflicts.[3] Automatic resolution of directory conflicts, for example, is well understood.

Within a partition, synchronization is maintained through the *CSS protocol*. Each partition chooses a Control and Synchronization Site (CSS) for each file group available to it. Files are locked at the CSS by Storage Sites (SS), and held at the SS's by Using Sites (US).

The high-level protocols of LOCUS assume that the underlying network is fully connected. By this we mean that if site A can communicate with site B, and site B with site C, then site A can communicate with site C. In practice, this may be done by routing messages from A to C through B, although the present implementation of LOCUS runs on a broadcast network where this is unnecessary. The assumption of transitivity of communication significantly simplifies the high-level protocols used in LOCUS.

The low-level protocols enforce that network transitivity. Network information is kept internally in both a high-level status table and a collection of virtual circuits.[2] The two structures are, to some extent, independent. Membership in the partition does not guarantee the existence of a virtual circuit, nor does an open virtual circuit guarantee membership in the partition. Failure of a virtual circuit, either on or after open, does, however, remove a node from a partition. Likewise removal from a partition closes all relevant virtual circuits. All changes in partitions invoke the protocols discussed later in this paper.

The system attempts to maintain file access across partition changes. If it is possible, without loss of information, to substitute a different copy of a file for one lost because of partition, the system will do so. If, in particular, a process loses contact with a file it was reading remotely, the system will attempt to reopen a different copy of the same version of the file.

The ability to mount filegroups independently gives great flexibility to the name space. Since radical changes to the name space can confuse users, however, this facility is rarely used for that purpose, and that use is not supported in LOCUS. The reconfiguration protocols require that the mount hierarchy be the same at all sites.

4 Requirements for the Reconfiguration Protocols

The first constraint on the reconfiguration protocol is that it maintain consistency with respect to the internal system protocols. All solutions satisfying this constraint could be termed correct. Correctness, however, is not enough. In addition to maintaining system integrity, the solution must insulate the user from the underlying system changes. The solution should not affect program development, and it should be efficient enough that any delays it imposes are negligible.

[1] An elegant strategy for detecting conflicts can be found [Parker80].

[2] The virtual circuits deliver messages from *site* A to *site* B (the virtual circuits connect sites, not processes) in the order they are sent. If a message is lost, the circuit is closed. The mechanism defends the local site from the slow operation of a foreign site.

As an example of a "correct" but poor solution to the problem, the O.S. could handle only the boot case, where all machines in the network come up together. Any failures would be handled by a complete network reboot. Such a solution would easily satisfy the consistency constraint; however, one might expect murmurs of complaint from the user community.

Similarly, a solution that brings the system to a grinding halt for an unpredictable length of time at unforseeable intervals to reconstruct internal tables might meet the requirement of correctness, but would clearly be undesirable.

Optimally, the reconfiguration algorithms should not affect the user in any matter whatsoever. A user accessing resources on machine A from machine B should not be affected by any activity involving machine C. This intuitive idea can be expressed in several principles:

1. User activity should be allowed to continue without adverse affect, provided no resources are lost.
2. Any delay imposed by the system on user activity during reconfiguration should be negligible.
3. The user should be shielded from any transient effects of the network configuration.
4. Any activity initiated after the reconfiguration should reflect the state of the system after the reconfiguration.
5. Specialized users should be able to detect reconfigurations if necessary.
6. No user should be penalized for increased availability of resources.[1]

All these principles are fairly intuitive. They merely extend the concept of network transparency to a dynamic network and express a desire for efficiency. They do, however, give tight constraints on the eventual algorithms. For example, those operations with high delay potentials must be partitioned in such a way that the tasks relevant to a specific user request can be run quickly, efficiently, and immediately.

The principles have far-reaching implications in areas such as file access and synchronization. Suppose, for example, a process were reading from a file replicated twice in its partition. If it were to lose contact with the copy it was reading, the system should substitute the other copy (assuming, of course, that it is still available). If a more recent version became available, the process should continue accessing the old version, but this must not prevent other processes from accessing the newer version.

These considerations apply equally to all partitions, and no process should loose access to files simply because a merge occurred. While the LOCUS protocols insure synchronization within a partition, they cannot do so between partitions. Thus, it is easy to contrive a scenario where the system must support conflicting locks within a single partition, and invoke any routines necessary to deal with inconsistencies that result.

[1] This last point may cause violations of synchronization policies, as discussed below.

5 Protocol Structure

As noted before, the underlying LOCUS protocols assume a fully-connected network. To insure correct operation, the reconfiguration strategy must guarantee this property. If, for instance, a momentary break occurs between two sites, all other sites in the partition must be notified of the break. A simple scan of available nodes is insufficient.

The present strategy splits the reconfiguration into two stages: first, a *partition* protocol runs to find fully-connected sub-networks; then a *merge* protocol runs to merge several such sub-networks into a full partition. The partition protocol affects only those sites previously thought to be up. It divides a partition into sub-partitions, each of which is guaranteed to be fully-connected and disjoint from all other sub-partitions. It detects all site and communications failures and cleans up all affected multi-site data structures, so that the merge protocol can ignore such matters. The merge protocol polls the set of available sites, and merges several disjoint sub-partitions into one.

After the new partition is established, the recovery procedure corrects any inconsistencies brought about either by the reconfiguration code itself, or by activity while the network was not connected. Recovery is concerned mainly with file consistency. It schedules update propagation, detects conflicts, and resolves conflicts on classes of files it recognizes.

All reconfiguration protocols are controlled by a high-priority kernel process. The partition and merge protocols are run directly by that process, while the recovery procedure runs as a privileged application program.

6 The Partition Protocol

Communication in a fully-connected network is an equivalence relation. Thus the partitions we speak about are partitions, in the strict mathematical, sense of the set of nodes of the network. In normal operation, the site tables reflect the equivalence classes: all members of a partition agree on the status of the general network. When a communication break occurs, for whatever reason, these tables become unsynchronized. The partition code must re-establish the logical partitioning that the operating system assumes, and synchronize the site tables of its member sites to reflect the new environment.

In general, a communication failure between any two sites does not imply a failure of either site. Failures caused by transmission noise or unforeseen delays cannot be detected directly by foreign sites, and will often be detected in only one of the sites involved. In such situations, the partition algorithm should find maximum partitions: a single communications failure should not result in the network breaking into three or more parts.[1] LOCUS implements a solution based on iterative intersection.

[1] Breaking a virtual circuit between two sites aborts any ongoing activity between those two sites. Partition fragmentation must be minimized to minimize the loss of work.

A few terms are helpful for the following discussion. The *partition set*, P_α, is the set of sites believed up by site α. The *new partition set*, P'_α, is the set of sites known by α to have joined the new partition.

Consider a partition P after some set of failures has occurred. To form a new partition, the sites must reach a consensus on the state of the network. The criterion for consensus may be stated in set notation as: for every $\alpha,\beta \in P$, $P_\alpha = P_\beta$. This state can be reached from any initial condition by taking successive intersections of the partition sets of a group of sites.

When a site α runs the partition algorithm, it polls the sites in P_α. Each sites polled responds with its own partition set $P_{pollsite}$. When a site is polled successfully, it is added to the new partition set P'_α, and P_α is changed to $P_\alpha \cap P_{pollsite}$. α continues to poll those sites in P_α but not in P'_α until the two sets are equal, at which point a consensus is assured, and α announces it to the other sites.

Translating this algorithm into a working protocol requires provisions for synchronization and failure recovery. These two requirements are antagonistic--while the algorithm requires that only one active site poll for a new partition, and that other sites join only one new partition, reliability coniderations require that sites be able to change active sites when one fails--and make the protocol intrinsically complex. Since the details of the protocol are not relevant to the overall discussion, they have been placed in an appendix.

7 The Merge Protocol

The *merge* procedure joins several partitions into one. It establishes new site and mount tables, and re-establishes CSS's for all the file groups. To form the largest possible partition, the protocol must check all possible sites, including, of course, those thought to be down. In a large network, sequential polling results in a large additive delay because of the timeouts and retransmissions necessary to determine the status of the various sites. To minimize this effect, the merge strategy polls the sites asynchronously.

The algorithm itself is simple. The site initiating the protocol sends a request for information to all site in the network. Those sites which are able respond with the information necessary for the initiating site to build the global tables. After a suitable time, the initiating site gives up on the other sites, declares a new partition, and broadcasts its composition to the world.

The algorithm is centralized and can only be run at one site, and a site can only participate in one protocol at a time, so the other sites must be able to halt execution of the protocol. To accomplish this, the polled site sends back an error message instead of a normal reply:

```
IF ready to merge THEN
  IF merging AND actsite == locsite THEN
    IF fsite < locsite THEN
      actsite := fsite;
      halt active merge;
      OK
    ELSE
```

```
      NO
     FI
    ELSE
     actsite := fsite;
     OK
    FI
  ELSE
    NO
  FI
```

If a site is not ready to merge, then either it or some other site will eventually run the merge protocol.

The major source of delay in the merge procedure is in the timeout routines that decide when the full partition has answered. A fixed length timeout long enough to handle a sizeable network would add unreasonable delay to a smaller network or a small partition of a large network. The strategy used must be flexible enough to handle the large partition case and the small partition case at the same time.

The merge protocol waits longer when there is a reasonable expectation that further replies will arrive. When a site answers the poll, it sends its partition information in the reply. Until all sites believed up by some site in the new partition have replied, the timeout is long. Once all such sites have replied, the timeout is short.

8 The Recovery Procedure

Even before the partition has been reestablished, there is considerable work that each node can do to clean up its internal data structures. Essentially, each machine, once it has decided that a particular site is unavailable, must invoke failure handling for all resources which it's processes were using at that site, or for all local resources which processes at that site were using. The action to be taken depends on the nature of the resource and the actions that were under way when failure occurred. The cases are outlined in the table below.

Local Resource in Use Remotely

Resource	Failure Action
File (open for update)	Discard pages, close file and abort updates
File (open for read)	Close file

Remote Resource in Use Locally

Resource	Failure Action
File (open for update)	Discard pages, set error in local file descriptor
File (open for read)	Internal close, attempt to reopen at other site

Interacting Processes

Failure Type	Action
Remote Fork/Exec, remote site fails	return error to caller
Fork/Exec, calling site fails	abort local process
Distributed Transaction	abort all related subtransactions in partition

Once the machines in a partition have mutually agreed upon the membership of the partition, there is still additional work to be done in the file system in order to make the internal data structures consistent again. LOCUS permits a given file to be updated in different partitions, so that one must determine whether, for each file now available, a conflict has occurred, i.e. uncoordinated update has actually occurred at different sites during partition. In those cases where it has, if the system is able, automatic reconciliation must be done. Directories are an important case of a type of file where this action can be taken. See [Popek83a] for a discussion of how these steps are taken. Finally, if there are operations in progress which would not be permitted during normal behavior, some action must be taken. For example, file X is open for update in two partitions, the system policy permits only one such use at a time, and a merge occurs. The desired action is to permit these operations to continue to completion, and only then perform file system conflict analysis on those resources.[1] Lastly, the system must select, for each group of files it supports, a new synchronization site. This is the site to which the LOCUS file system protocols direct all file open requests. See [Walker83] for a discussion of the normal operation protocols for the file system. Once the synchronization site has been selected, that site must reconstruct the lock table for all open files from the information remaining in the partition.

After all these functions have been completed, the effect of topology change has been completely processed. For most of these steps, normal processing at all of the operating nodes continues unaffected. If a request is made for a resource which has not been merged yet, the normal order of processing is set aside to handle that request. Therefore, higher level reconfiguration steps, such as file and directory merge, do not significantly delay user requests.

9 Protocol Synchronization

The reconfiguration procedure breaks down into three distinct components, each of which has already been discussed. What remains is a discussion of how the individual parts are tied together into a robust whole. At various points in the procedure, the participating sites must be synchronized, and control of the protocol must be handed to a centralized site. Those sites not directly involved in the activity must be able to ascertain the status of the active sites to insure that no failures have stalled the entire network.

One approach to synchronization would be to add ACKs to the end of each section of the protocol, and get the participants in lock-step before proceeding to the next section. This approach increases both the message traffic and the delay, both critical performance quantities. It also requires careful analysis of the critical sections in the protocols to determine where a commit is required, and the implementation of a commit for each of those sections. If a site fails during a synchronization stage, the system must still detect and recover from that failure.

[1] LOCUS currently does not support this behavior.

LOCUS reconfiguration uses an extension of a "failure detection" mechanism for synchronization control. Whenever a site takes on a passive role in a protocol, it checks periodically on the active site. If the active site fails, the passive site can restart the protocol.

As the various protocols execute, the states of both the active and the passive sites change. An active site at one instant may well become a passive site the next, and a passive site could easily end up waiting for another passive site. Without adequate control, this could lead to circular waits and deadlocks.

One solution would be to have passive sites respond to the checks by returning the site that they themselves are waiting for. The checking site would then follow that chain and make sure that it terminated. This approach could require several messages per check, however, and communications delays could make the information collected obsolete or misleading.

Another alternative, the one used in LOCUS, is to order all the stages of the protocol. When a site checks another site, that site returns its own status information. A site can wait only for those sites who are executing a portion of the protocol that preceeds its own. If the two sites are in the same state, the ordering is by site number. This ordering of the sites is complete. The lowest ordered site has no site to legally wait for; if it is not active, its check will fail, and the protocol can be re-started at a reasonable point.

While no synchronization "failures" can cause the protocols to fail, they can slow execution. Without ACKs, the active site cannot effectively push its dependents ahead of itself through the stages of the protocol. Nor can it insure that two passive sites always agree on the present status of the reconfiguration. On the other hand, careful design of the message sequences can keep the windows where difficulties can occur small, and the normal case executes rapidly.

10 Future Work

Problems yet to be addressed include: a) selective operation of protocols--at present all protocols are run at every configuration change; b) optimization of recovery strategies--recovery is now ignorant of recent history and makes more general checks than are sometimes necessary; and c) automatic reconciliation of multifile objects. While a much work has been done on multi-file objects, [Mueller83][Faissol81][Popek83a], the basic unit of synchronization in LOCUS is the file. A more general synchronization mechanism could simplify both conflict detection and resolution.

11 Conclusions

The difficulties involved in dynamically reconfiguring an operating system are both intrinsic to the problem, and dependent on the particular system. Rebuilding lock tables and synchronizing processes running in separate environments are problems of inherent difficulty. Most of the system-dependent problems can be avoided, however, with careful design.

The fact that LOCUS uses specialized protocols for operating system to operating system communication made it possible to control message traffic quite selectively. The ability to alter specific protocols to simplify the reconfiguration solution was particularly appreciated.

The task of developing a protocol by which sites would agree about the membership of a partition proved to be surprisingly difficult. Balancing the needs of protocol synchronization and failure detection while maintaining good performance presented a considerable challenge. Since reconfiguration software is run precisely when the network is flaky, those problems are real, and not events that are unlikely.

Nevertheless, it has been possible to design and implement a solution that exhibits reasonably high performance. Further work is still needed to assure that scaling to a large network will successfully maintain that performance characteristic, but our experience with the present solution makes us quite optimistic.

12 Bibliography

[Walker83] Walker, B., and G. J. Popek, "The LOCUS Distributed File System", submitted to 9th SOSP.

[Parker80] Parker, D. Stott, Popek, Gerald J., Rudisin, G., Stoughton, A., Walker, B., Walton, E., Chow, J., Edwards, D., Kiser, S., and Kline, C., "Detection of Mutual Inconsistency in Distributed Systems", accepted for publication in IEEE Transactions of Software Engineering, to appear May 1983.

[Popek83] Popek, G., and B. Walker, "Network Transparency and its Limits in a Distributed Operating System", submitted to 9th SOSP.

[Mueller83] Mueller, E., J. Moore, and G. Popek, "A Nested Transaction System for LOCUS", submitted to IFIPS Congress.

[Popek83a] Popek, G., and Thiel, G., "Recovery of Replicated Storage in Distributed Systems", submitted to the 9th SOSP.

[Faissol81] Faissol, S., "Availability and Reliability Issues in Distributed Databases", Ph. D. Dissertation, Computer Science Department, University of California, Los Angeles, 1981.

[Millstein 77] Millstein, R., "The National Software Works: A Distributed Processing System", Proceedings of the ACM National Conference, (1977).

[Birell82] Birrell, A.D., R. Levin, R. M. Needham, and M.D. Schroeder, "Grapevine: An Exercise in Distributed Computing", Communications of the ACM, 25:4, April 1982.

[Ousterhout82] Ousterhout, J.K., "Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa", CMU Technical Report CMU-CS-80-112, April 1980

13 Appendix A: Partition Protocol Details

To satisfy the dual requirements of synchronization and reliability, the partition protocol exploits several regularities based on the natural ordering of sites in the network by site number. For instance, the new partition set, rather than being independent of the current partition set, is the subset of the current partition set with site numbers less than the point of polling. At the active site, this is a dynamic quantity, but at the passive sites it is their own site number (all sites lower than them in site number have already been polled by the active site).

A site, α, initiates the protocol whenever it notices a network failure. The site records the failure by removing the offending site from its local partition set P_α. The site then scans the remaining sites in P_α in ascending order until it finds a site β willing to join its partition.[1] β sets its partition set P_β to $P_\alpha \cap P_\beta$ and assumes the task of polling. At this point, however, only β has joined a partition; α will not join a partition until β polls it.

At this stage β is guaranteed to be the lowest numbered site in its partition set. It continues scanning sites in ascending order. When β has scanned all sites in P_β, the new partition, now defined, is sent to all sites in the *old* partition (so that they may begin their own scanning when necessary). The algorithm for polling is

```
FOR fsite FROM 1 TO locsite-1 WHILE UNTIL inpart OR actsite != NULL
 DO
  IF oktojoin (fsite) THEN
   $P_{locsite} := P_{locsite} \cap P_{fsite}$;
   actsite := fsite
  ELSE
   $P_{locsite} := P_{locsite} - P'_{fsite}$
  FI
 OD
IF actsite == NULL THEN actsite := locsite; inpart := TRUE FI
IF actsite == locsite THEN
 FOR fsite FROM locsite+1 TO nsite
  DO
   IF oktojoin (fsite) THEN
    $P_{locsite} := P_{locsite} \cap P_{fsite}$
   ELSE
    $P_{locsite} := P_{locsite} - P'_{fsite}$
   FI
  OD
ELSE
 config_wait
FI
```

The subroutine *oktojoin* polls a site, asking if it will join the new partition. It returns

[1] A site is always willing to join its own partition. If the first site α successfully scans is itself, $\alpha = \beta$.

true if the foreign site will join, and false if it refuses, or is inaccessible. *inpart* is a boolean flag that shows whether the local site has joined a partition.

If the site starts execution of this algorithm due to a locally recognized condition, neither inpart nor actsite will be set. The site will poll all sites less than itself attempting to find an active site. If it succeeds, that site becomes the active site and the local site waits for it to finish the protocol. If it fails, then it becomes the active site and polls the sites with site number greater than itself.

If the site begins execution due to action by a foreign site, then inpart and actsite will both be set. If the active site is foreign, the local site waits for completion of the protocol. If it is local, the order of polling assures that all sites lower in number have already been polled, and the site polls only the higher numbered sites.

The polling algorithm is straightforward. The need to respond reasonably to site failures complicates the decision algorithm at the polled site somewhat, as a passive site can never be sure of the state of its corresponding active site. Suppose we based our decision solely on the basis spelled out in the algorithm:

```
IF NOT inpart AND $P'_{fsite} \subset P_{locsite}$ THEN
  YES
ELSE
  NO
FI
```

Once a site joined a partition, it could join no other. If the active site became separated, for whatever reason, the sites that had already joined its partition would hang--waiting for Godot, as it were. Adding a periodic check of the controlling site helps, but is not sufficient. The sites would no longer hang, but the resulting partition could be highly fragmented: as the sites notice, independently, that the active site has vanished, they start the algorithm themselves. None of the sites still waiting would join the new partition, believing they are already part of one. In the worst case, this could lead to total network fragmentation.

The polled site needs to detect these conditions when they occur. On possible way to do so is to check the active site whenever asked to join a partition different from the current one. This, however, leads to unnecessary message traffic and potentially serious timing problems. Far better would be a method allowing a site to decide solely on the basis its internal information and the information sent by the foreign site.

There are two cases to deal with. If the polling site has a lower site number than the polled site, the normal decision criterion works. Here $fsite \in P'_{locsite}$, which implies $fsite \in P'_{actsite}$. Since fsite is no longer in $P'_{actsite}$ (else it would not be polling), the partition being built by the active site is wrong even if it does still exist, and the polled site can safely join another.

The second case, where the polling is downward, is more difficult, there being no reliable way for the lower site to determine whether the polling site was a member of

P_{active} if the lower site is not the active site.[1] Without this determination, the lower site cannot join the partition of the polling site. Prevention of fragmentation must occur at a higher level.

With this in mind, here is the algorithm used at the polled site:

```
IF $P'_{fsite} \subset P_{locsite}$ THEN
  IF fsite < locsite THEN
    actsite := fsite;
    $P_{locsite} := P_{locsite} \cap P_{fsite}$;
    reply := OK
  ELIF inpart THEN
    IF actsite = locsite THEN
      $P_{locsite} := P_{locsite} \cap P_{fsite}$;
      reply := OK
    ELSE
      reply := NO
    FI
  ELSE
    actsite := locsite;
    $P_{locsite} := P_{locsite} \cap P_{fsite}$;
    reply := OK
  FI
ELIF actsite = locsite AND
 (*)   $fsite \in P'_{locsite}$ THEN
  restart partition algorithm;
  $P_{locsite} := P_{locsite} \cap P_{fsite}$;
  reply := OK
ELSE
  reply := NO
FI
```

The first test checks the local site's eligibility to join the new partition. Subsequent tests in that section determine the status of the local site vis-a-vis other partitions which may be forming. The test marked (*) is a special case making use of the extra information available at the active site regarding the new partition set. If the polling site is a member of the active site's new partition set, it must have noticed the failure of a site in its own P', and started polling itself. This makes the partition set of the active site invalid, and it must restart the protocol.

The loss of an active site can still lead to the complete fragmentation of a partition if the sites detect the loss in descending order. The wait-probe-timeout strategy is designed to prevent just such an occurence. The sites joining a partition probe only the sites immediately beneath them. If they declare a site down, they leave the partition and begin scanning for a new one *beginning with the active site of the previous partition*. The test marked (*) in the above algorithm deals with just that eventuality. As noted above, such a poll causes the old active site to restart its poll. Fragmentation can only occur when a set of sites including the active site leaves the partition, and

[1] Again, for efficiency reasons, we do not consider the possibility of polling the active site.

then the number of fragments resulting cannot exceed the number of sites lost.[2] It may be possible to guarantee maximum partitions, but doing so requires a considerable amount of message traffic and additional complexity. Since the probability of active site failure is expected to be low, particularly in conjunction with secondary site failures, and the cost of the breakdown noted is not high, this solution is sufficient.

[1] In particular, if only the active site is lost, the number of fragments will be one: the entire partition.

REFERENCES

DOWN77 Downs, D., and Popek, G. J., "A kernel design for a secure data base management system," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan, October, 1977, 507-514.

DOWN79 Downs, D., and Popek, G., "Data base system security and Ingres," Proceedings of the conference on Very Large Data Bases, 1979, Rio De Janiero.

KAMP77 Kampe, M., Kline, C. S., Popek, G. J., and Walton, E. J., The UCLA Data Secure Operating System Prototype, Computer Science Department, University of California, Los Angeles, Technical Report 77-3, July 1977.

KEMM78 Kemmerer, R. A., A proposal for the formal verification of the security properties of the UCLA Secure UNIX Operating System Kernel, Computer Science Department, University of California, Los Angeles, Technical Report 78-1 (UCLA-ENG-7810), February 1978.

KEMM79 Kemmerer, R. A., Formal verification of the UCLA Security Kernel: Abstract model, mapping functions, theorem generation, and proofs, Ph.D. thesis, UCLA-ENG-7956, University of California at Los Angeles, 1979.

KLIN77 Kline, C. S., and Popek, G. J., Encryption in computer network security, Computer Science Department, University of California, Los Angeles, Technical Report 77-2, April 1977.

KLIN79 Kline, C. S., and Popek, G. J., "Public key vs. conventional key encryption", Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 831-837.

MENA79 Menasce, D. A., Rudisin, G. J., Popek, G. J., and Kline, C. S., A proposed architecture for the distributed secure system base, Computer Science Department, University of California, Los Angeles, Technical Report 79-10 (UCLA-ENG-7957), September 1979.

POPE73 Popek, G. J., "Correctness in access control," Proceedings of the ACM National Conference, Atlanta, Georgia, 1973, 236-241.

POPE74a Popek, G. J., "Protection structures," IEEE Computer, (Jul. 1974), 22-33.

POPE74b Popek, G. J., "A principle of kernel design," Proceedings of the National Computer Conference, 43 (1974), AFIPS Press, Arlington, Va., 977-978.

POPE74c Popek, G. J., and Kline, C. S., "Verifiable secure operating system software," Proceedings of the National Computer Conference, 43 (1974), AFIPS Press, Arlington, Va., 145-151.

POPE74d Popek, G. J., and Kline, C. S., "The design of a verified protection system," Proceedings of the IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August, 1974, 183-196.

POPE75a Popek, G. J., and Kline, C. S., "A verifiable protection system," Proceedings of the 1975 International Conference on Reliable Software, Los Angeles, Ca., April 21-23, 1975, 294-304.

POPE75b Popek, G. J., and Kline, C. S., "The PDP-11 virtual machine architecture: a case study," Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, Texas, October 1975.

POPE76 Popek, G. J., and Farber, D. A., "On computer security verification," Twelfth IEEE Computer Society International Conference: Compcon 76, San Francisco, Ca., Feb. 1976, 140-145.

POPE77a Popek, G. J., Horning, J. J., Lampson, B. W., Mitchell, J. G., and London, R. L., "Notes on the design of EUCLID," Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March, 1977, 11-18. Also in SIGPLAN Notices, 12, 3 (Sept. 1977).

POPE77b Popek, G. J., and Kline, C. S., "Encryption protocols, public key algorithms and digital signatures in computer networks," Foundations of Secure Computing, R. DeMillo, et. al., eds., Academic Press, New York, 1978, 133-153.

POPE78a Popek, G. J., and Kline, C. S., "Design issues for secure computer networks", Operating Systems, An Advanced Course, R. Bayer, R. M. Graham, G. Seegmuller, Eds., Springer-Verlag, New York, 1978

POPE78b Popek, G. J., and Farber, D. A., "A model for verification of data security in operating systems," Communications of the ACM, 21, 9 (Sept. 1978), 737-749.

POPE78c Popek, G. J., and Kline, C. S., "Issues in kernel design," Proceedings of the National Computer Conference, 47 (1978), AFIPS Press, Arlington, Va., 1079-1086.

POPE78d Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A. H., Urban, M. P., and Walton, E. J., UCLA Data Secure Unix -- a secure operating system: software architecture, Technical Report 78-7 (UCLA-ENG-7854), Computer Science Department, University of California, Los Angeles, 1978.

POPE79 Popek, G. J., Kampe, M., Kline, C. S., Stoughton, A., Urban, M., and Walton, E. J., "UCLA Secure Unix," Proceedings of the National Computer Conference, 48 (1979), AFIPS Press, Arlington, Va., 355-364.

URBA79 Urban, M. P., The design and implementation of a file policy manager for the UCLA Data Secure Unix System, M.S. Thesis, Computer Science Department, University of California, Los Angeles, 1979.

WALK77 Walker, B. J., Verification of the UCLA Security Kernel: Data Defined Specifications, M. S. Thesis, Computer Science Department, University of California, Los Angeles, 1977.

WALK80 Walker, B. J., Kemmerer, R. A., and Popek, G. J., "Specification and verification of the UCLA Unix Security Kernel," Communications of the ACM, 23, 2 (Feb. 1980), 118-131.

WALT75 Walton, E. J., The UCLA Security Kernel, M.S. Thesis, Computer Science Department, University of California, Los Angeles, June 1973.

BIBLIOGRAPHY

Popek, G.J. "Correctness in Access Control," Proceedings of the ACM National Conference, Atlanta, Georgia, August 1973, pp. 236-241.

Popek, G.J. and R.P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures," ACM/SIGOPS Fourth Symposium on Operating System Principles, Yorktown Heights, New York, October 15-17, 1973. Revised version published in Communications of the ACM, 17(7):412-421, July 1974.

Popek, G.J. "A Principle of Kernel Design," AFIPS Conference Proceedings, Vol. 43: 1974 National Computer Conference, Chicago, Illinois, May 6-10, 1974, AFIPS Press, Montvale, New Jersey, 1974, pp. 977-978.

Popek, G.J. and C.S. Kline. "Verifiable Secure Operating System Software," AFIPS Conference Poceedings, Vol. 43: 1974 National Computer Conference, Chicago, Illinois, May 6-10, 1974, AFIPS Press, Montvale, New Jersey, 1974, pp. 145-151.

Popek, G.J. "Protection Structures," IEEE Computer, June 1974, pp. 22-23.

Popek, G.J. "A Note on Secure Data Base Design," Computer Science Department, University of California, Los Angeles, Technical Report 74-4, July 1974.

Popek, G.J. and C.S. Kline. "The Design of a Verified Protection System," Proceedings of the IRIA International Workshop on Protection in Operating Systems, Rocquencourt, France, August 13-14, 1974, pp. 183-196.

Snuggs, M.A.L., G.J. Popek, and R.J. Peterson. "Data Base System Objectives as Design Constraints," Proceedings of the ACM National Conference, San Diego, California, November 1974, pp. 641-647.

Popek, G.J. "On Data Secure Computer Networks," Proceedings of the ACM SIGCOMM/SIGARCH Workshop on Network Communications, Santa Monica, California, March 1975, pp. 59-62.

Popek, G.J. and C.S. Kline. "A Verifiable Protection System," Proceedings of the 1975 International Conference on Reliable Software, Los Angeles, California, April 21-23, 1975, pp. 294-304.

Walton, E.J. The UCLA Security Kernel, M.S. in Computer Science, Computer Science Department, University of California, Los Angeles, June 1975.

Popek, G.J. "On the Current State of Protection in Computer Systems," Eleventh IEEE Computer Society Conference: Compcon '75 Fall, Washington, D.C., September 9-11, 1975, pp. 40-41.

Popek, G.J. and C.S. Kline. "The PDP-11 Virtual Machine Architecture: A Case Study," Proceedings of the Fifth Symposium on Operating Systems Principles, Austin, Texas, October 1975.

Walton, E.J. "The UCLA Pascal Translation System," Computer Science Department, University of California, Los Angeles, UCLA-ENG-7954 (DAHC-73-C-0368), January 1976.

Popek, G.J. and D.A. Farber. "On Computer Security Verification," Twelfth IEEE Computer Society International Conference: Compcon '76 Spring, San Francisco, California, February 24-26, 1976, pp. 140-145.

Farber, D.A. "A Model of Program Translation," Computer Science Department, University of California, Los Angeles, Technical Report 76-3, August 1976.

Popek, G.J., J.J. Horning, B.W. Lampson, R.L. London, and J.G. Mitchell. "The Programming Language Euclid," Computer Science Department, University of California, Los Angeles, Technical Report 76-4, September 1976.

Walker, B.J. "Note on Proof of Concrete Code," Computer Science Department, University of California, Los Angeles, Technical Report 76-5, September 1976.

Abraham, S.M. A Protection Design for the UCLA Security Kernel, M.S. in Computer Science, Computer Science Department, University of California, Los Angeles, December 1976.

Popek, G.J., J.J. Horning, B.W. Lampson, J.G. Mitchell, and R.L. London. "Notes on the Design of EUCLID," Proceedings of an ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March 28-30, 1977, ACM, Inc., New York, New York, 1977, pp. 11-18. Also published in SIGPLAN Notices, 12(3):11-18, 1977.

Kline, C.S. and G.J. Popek. "Encryption in Computer Network Security," Computer Science Department, University of California, Los Angeles, Technical Report 77-2, April 1977.

Kampe, M., C.S. Kline, G.J. Popek, and E.J. Walton. "The UCLA Data Secure Operating System Prototype," Computer Science Department, University of California, Los Angeles, Technical Report 77-3, July 1977.

Popek, G.J. and C.S. Kline. "Design Issues for Secure Computer Networks," presented at the Advanced Course in Operating Systems, July 28-August 5, 1977 and March 29-April 6, 1978, Munich, Germany. In: Operating Systems, An Advanced Course, edited by R. Bayer et al., Springer-Verlag, Berlin, Germany, 1978, pp. 518-546 (Lecture Notes in Computer Science, Vol. 60, edited by G. Goos and J. Hartmanis).

Popek, G.J. and C.S. Kline. "Issues in Kernel Design," presented at the Advanced Course in Operating Systems, July 28-August 5, 1977 and March 29-April 6, 1978, Munich, Germany. In: Operating Systems, An Advanced Course, edited by R. Bayer et al., Springer-Verlag, Berlin, Germany, 1978, pp. 209-226 (Lecture Notes in Computer Science, Vol. 60, edited by G. Goos and J. Hartmanis). Also published in AFIPS Conference Proceedings, Vol. 47: 1978 National Computer Conference, Anaheim, California, June 5-8, 1978, AFIPS Press, Montvale, New Jersey, 1978, pp. 1079-1086.

Menasce, D.A., G.J. Popek, and R.R. Muntz. "A Locking Protocol for Resource Coordination in Distributed Systems," Computer Science Department, University of California, Los Angeles, Technical Report 77-6 (UCLA-ENG-7808), October 1977.

Popek, G.J. and C.S. Kline. "Encryption Protocols, Public Key Algorithms and Digital Signatures in Computer Networks," Proceedings of the Atlanta Conference on Fundamentals of Secure Computing, Atlanta, Georgia, October 2-5, 1977. Also published in Foundations of Secure Computation, edited by R.A. de Millo, et al., Academic Press, Inc., New York City, New York, 1978, pp. 133-153.

Downs, D. and G.J. Popek. "A Kernel Design for a Secure Data Base Management System," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan, October 6-8, 1977, IEEE, 1977, pp. 507-514. Also published in Data Base Engineering, 1(4):8-15, December 1977.

Walker, B.J. "Verification of the UCLA Security Kernel: Data Defined Specifications," Computer Science Department, University of California, Los Angeles, Technical Report 77-9 (UCLA-ENG-7809), November 1977 (Also published as a M.S. Thesis).

Kemmerer, R.A. "A Proposal for the Formal Verification of the Security Properties of the UCLA Secure UNIX Operating System Kernel," Computer Science Department, University of California, Los Angeles, Technical Report 78-1 (UCLA-ENG-7810), February 1978.

Popek, G.J. "Secure Distributed Processing Systems: Quarterly Management Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-2, April 30, 1978.

Menasce, D.A., G.J. Popek, and R.R. Muntz. "A Locking Protocol for Resource Coordination in Distributed Databases," Proceedings of the 1978 ACM/SIGMOD International Conference on Management of Data, Austin, Texas, May 31-June 2, 1978 (extended abstract). Full paper published in Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, August 29-31, 1978. Also published in ACM Transactions on Database Systems, 5(2):103-138, June 1980.

Popek, G.J. "Security in Network Operating Systems: A Survey," report prepared for the Institute for Computer Sciences and Technology, National Bureau of Standards, June 30, 1978.

Popek, G.J. "Secure Distributed Processing Systems: Quarterly Technical Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-5 (UCLA-ENG-7853), June 1978.

Badal, D.Z. and G.J. Popek. "A Proposal for Distributed Concurrency Control for Partially Replicated Distributed Database Systems," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, August 29-31, 1978, pp. 273-286.

Menasce, D.A. and R.R. Muntz. "Locking and Deadlock Detection in Distributed Databases," Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, August 29-31, 1978, pp. 215-232. Also published in IEEE Transactions on Software Engineering, 5(3):195-202, May 1979.

Popek, G.J., M. Kampe, C.S. Kline, A.H. Stoughton, M.P. Urban, and E.J. Walton. "UCLA Data Secure UNIX - A Secure Operating System: Software Architecture," Computer Science Department, University of California, Los Angeles, Technical Report 78-7 (UCLA-ENG-7854), August 1978.

Popek, G.J. and D.A. Farber. "A Model for Verification of Data Security in Operating Systems," Communications of the ACM, 21(9):737-749, September 1978.

Menasce, D.A., G.J. Popek, and R.R. Muntz. "Centralized and Hierarchical Locking in Distributed Databases," IEEE Computer Society International Computer Software and Applications Conference: Distributed Data Base Management (Tutorial), Chicago, Illinois, November 1978, IEEE, New York City, New York, 1978, pp. 178-195.

Popek, G. J. "Secure Distributed Processing Systems: Quarterly Technical Report," Computer Science Department, University of California, Los Angeles, Technical Report 78-13 (UCLA-ENG-7955), December 1978.

Menasce, D. A., R. R. Muntz, and G. J. Popek. "A Formal Model of Crash Recovery in Computer Systems," Proceedings of the Twelfth Hawaii International Conference on System Sciences, Vol. I: Selected Papers in Software Engineering and Mini-Micro Systems, Honolulu, Hawaii, January 4-5, 1979, Western Periodicals Company, New York, 1979, pp. 28-35.

Chiappa, N. and A. H. Stoughton. "Programming the Local Network Interface," MIT Laboratory for Computer Science, Network Implementation Note No. 2, January 12, 1979.

Urban, M. P. The Design and Implementation of a File Policy Manager for the UCLA Data Secure Unix System, M. S. in Computer Science, Computer Science Department, University of California, Los Angeles, March 1979.

Badal, D. Z. and G. J. Popek. "Cost and Performance Analysis of Semantic Integrity Validation Methods," Proceedings of the 1979 ACM/SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30-June 1, 1979.

Kline, C. S. and G. J. Popek. "Public Key vs. Conventional Key Encryption," AFIPS Conference Proceedings, Vol. 48: 1979 National Computer Conference, New York City, New York, June 4-7, 1979, AFIPS Press, Montvale, New Jersey, 1979, pp. 831-837.

Popek, G. J., M. Kampe, C. S. Kline, A. H. Stoughton, M. P. Urban, and E. J. Walton. "UCLA Secure UNIX," AFIPS Conference Proceedings, Vol. 48: 1979 National Computer Conference, New York City, New York, June 4-7, 1979, AFIPS Press, Montvale, New Jersey, 1979, pp. 355-364.

Badal, D. Z. Semantic Integrity, Consistency and Concurrency Control in Distributed Databases, Ph. D. in Computer Science, Computer Science Department, University of California, Los Angeles, June 1979.

Popek, G. J. "Secure Distributed Processing Systems: Quarterly Management Report," Computer Science Department, University of California, Los Angeles, Technical Report 79-8, June 30, 1979.

Popek, G. J. "Secure Distributed Processing Systems: Quarterly Technical Report," Computer Science Department, University of California, Los Angeles, Technical Report 79-9 (UCLA-ENG-7956), June 30, 1979.

Badal, D.Z. "On Efficient Monitoring of Database Assertions in Distributed Databases," Proceedings of the Fourth Berkeley Conference on Distributed Data Management and Computer Networks, Berkeley, California, August 28-30, 1979, pp. 125-137.

Menasce, D.A., G.J. Rudisin, G.J. Popek, and C.S. Kline. "A Proposed Architecture for the Distributed Secure System Base," Computer Science Department, University of California, Los Angeles, Technical Report 79-10 (UCLA-ENG-7957), September 1979.

Downs, D. and G.J. Popek. "Data Base Management Systems Security and INGRES," Proceedings of the Fifth International Conference on Very Large Data Bases, Rio de Janeiro, Brazil, October 3-5, 1979, IEEE, 1979, pp. 280-290.

Walker, B.J., R.A. Kemmerer, and G.J. Popek. "Specification and Verification of the UCLA Security Kernel," Preprints of the Seventh Symposium on Operating Systems Principles, Pacific Grove, California, December 10-12, 1979, pp. 23-24 (extended abstract). Also published in Communications of the ACM, 23(2):118-131, February 1980.

Menasce, D.A. "Coordination in Distributed Systems: Concurrency, Crash Recovery and Database Synchronization," Computer Science Department, University of California, Los Angeles, Technical Report 79-14 (UCLA-ENG-7977), December 1979 (Also published as a Ph.D. Dissertation, December 1978).

Kemmerer, R.A. "Formal Verification of the UCLA Security Kernel: Abstract Model, Mapping Functions, Theorem Generation, and Proofs," Computer Science Department, University of California, Los Angeles, Technical Report 79-15 (UCLA-ENG-7978), December 1979 (Also published as a Ph.D. Dissertation, June 1979).

Popek, G.J. and C.S. Kline. "Encryption and Secure Computer Networks," ACM Computing Surveys, 11(4):331-356, December 1979.

Rudisin, G.J. Architectural Issues in a Reliable Distributed File System, Computer Science Department, University of California, Los Angeles, Technical Report 80-xx (UCLA-ENG-8014), April 1980 (Also published as a M.S. Thesis, March 1980).

Kline, C.S. Data Security: Operating Systems and Computer Networks, Computer Science Department, University of California, Los Angeles, Technical Report 80-2, October 1980 (Also published as a Ph.D. Dissertation, October 1980).

Parker, D.S., G.J. Popek, G. Rudisin, A. Stoughton, B. Walker, E.

Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. "Detection of Mutual Inconsistency in Distributed Systems," Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, February 3-5, 1981, pp. 172-184. Also accepted for publication in IEEE Transactions on Software Engineering.

Stoughton, A.H. "Access Flow: A Protection Model Which Integrates Access Control and Information Flow," submitted for publication to 1981 Symposium on Security and Privacy, Oakland, California, April 27-29, 1981.

Faissol, S.Z. Operation of Distributed Database Systems Under Network Partitions, Computer Science Department, University of California, Los Angeles, Technical Report 81-1, June 1981 (Also published as a Ph.D. Dissertation, June 1981).

Popek, G.J., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. "LOCUS: A Network Transparent, High Reliability Distributed System," submitted for publication to Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

BIBLIOGRAPHY (ADDENDUM)

1. Badal, D., and G. Popek, "A Proposal for Distributed Concurrency Control for Partially Redundant Distributed Data Base Systems", UCLA Technical Report, 1978, 13 pp.

2. Downs, D., and G. Popek, "A Kernel Design for a Secure Data Base Management System", Proceedings on Very Large Data Bases, Tokyo, October 1977, pp 507-514.

3. Kemmerer, D., "A Proposal for the Formal Verification of the Security Properties of the UCLA Secure Unix Operating System Kernel", UCLA Technical Report UCLA-ENG-7810, SDPS-78-001, 65 pp.

4. Menasce, D., and R. Muntz, "Locking and Deadlock Detection in Distributed Databases", UCLA Technical Report, 1978.

5. Menasce, D., G. Popek and R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Systems", ACM Transactions on Database Systems, (to appear).

6. Popek, G., and C. Kline, "Design Issues for Secure Computer Networks", Operating Systems, An Advanced Course, Springer-Verlay, Berlin, 1978, pp. 517-546.

7. Popek, G., and C. Kline, "Encryption Protocols, Public Key Algorithms and Digital Signatures in Computer Networks", Proceedings of the Conference on Fundamentals of Secure Computing, Atlanta, Ga., November 1977.

8. Popek, G., and C. Kline, "Issues in Kernel Design", 1978 National Computer Conference Proceedings, AFIPS Press, pp. 1079-1086.

9. Popek, G., C. Kline and E. Walton, "UCLA Secure Unix", UCLA Technical Report, February 1978, 20 pp.

10. Walker, B., "Verification of the UCLA Security Kernel: Data Defined Specifications", UCLA Technical Report UCLA-ENG-7809, SPDS-77-002, November 1977, 206 pp.