SDPS-78-002
June 1978

(1)

# SECURE DISTRIBUTED PROCESSING SYSTEMS:

# QUARTERLY TECHNICAL REPORTS

Gerald J. Popek
Principal Investigator



SECURE SYSTEMS AND SOFTWARE
ARCHITECTURE GROUP



**COMPUTER SCIENCE DEPARTMENT**

**School of Engineering and Applied Science**
**University of California**
**Los Angeles**

# SECURE DISTRIBUTED PROCESSING SYSTEMS

## QUARTERLY TECHNICAL REPORT

1 July 1977 - 30 April 1978

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 No.: MDA-903-77-C-0211
ARPA Order No.: 3396
Program Code No. 7P10

# SECURE DISTRIBUTED PROCESSING SYSTEMS

Advanced Research Projects Agency
Quarterly Technical Reports

July 1977 - March 1978

## Introduction

This technical report covers research carried out by the Secure Distributed Processing Systems group at UCLA, under ARPA Contract MDA-903-77-C-0211 during the three quarters in the period July 1, 1977 to March 31, 1978. Significant advances have been made on all four contracted tasks, namely network security, data management security, high availability secure information management, and UCLA secure system enhancements. Below, we describe that progress and point to the list of references which represent the published work resulting from this supported research. → to p. 4

## Task I - Network Security

A number of significant steps have been taken over the last three quarters. First, UCLA is participating 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. A BCR unit has been received at UCLA and checkout has begun. Coordination of UCLA's initial role as a server host in the BCR experiment, and subsequently potentially as a key distribution center, has been coordinated with other ARPA contractors.

A major portion of the effort in network security has been 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 has been developed for the UCLA Secure Operating System Prototype. The implementation is now being improved to integrate it into the complete system. The final prototype is scheduled to be used in the Navy's ACCAT Guard project later this year.

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 ap-

plicable to existing networks such as the ARPANET.

Several architectural design and analysis efforts have also been in progress during this period, reported in references 6 and 7. The first presents a general view of design issues in network security, and has been used by the Mitre Corp. in the development of their network security methods for military systems. The other challenges much of the work on public key encryption methods, and shows that all digital signature methods previously proposed suffer from serious flaws. A superior method is then outlined.

## Task II - Data Management Security

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 severely limit the amount of software which must operate securely. At UCLA, we have succeeded in developing a general kernel based architecture, 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 as reference 2, and in order to demonstrate its operational feasibility, the INGRES data management system is currently being altered to include the proposed kernel structures. An important result of this test implementation, besides demonstrating feasibility, is that our approach is retrofittable to existing systems. The savings in existing software can be enormous.

## Task III - High Availability Secure Information Management

A significant new direction of the research at UCLA has been concerned with reliability, availability, and security in distributed systems. The core of this effort is the development of a highly available, secure distributed system base that can run in an integrated fashion on local networks, utilize existing equipment, and provide a base on top of which one can easily install such applications as distributed data management systems, electronic office facilities, and the like. The base is to be entirely responsible for backup, recovery, security, and much of system management. It should be easily extensible in terms of hardware additions and deletions, all without software alterations or user knowledge.

The design of this system base has progressed a great deal

over the past three quarters, and a preliminary design document was reviewed by other internationally known researchers. A more complete design report will be completed this summer, at which time prototype development will commence.

In conjunction with this research direction, a number of other strong efforts have been completed. First is a complete protocol for coordination of resources in a distributed system, that permits arbitrary failures of nodes, links, and software modules, either during normal operation or recovery. Synchronization suitable for sophisticated data management is provided, and the correctness of the entire protocol is proven. The work is reported in reference 5, and has been accepted for publication in the top journal of the field.

Several other protocols have also been developed for coordination of resources and detection of deadlock. These are now undergoing refinement and have been submitted for publication.

To support the distributed system development, UCLA has participated in the development of the Local Network Interfaces (LNIs) principally developed by UCI and MIT, and three Interfaces are scheduled to be installed at UCLA this summer, creating a three node network for use in development and measurement experiments.

Task IV - UCLA Secure System Enhancement

The UCLA Secure system prototype has been the focus of considerable progress. First, a great deal of energy has been spent in completing the prototype. The basic task is now essentially done. Full function operation has been demonstrated. Fine grained protection on an individual file basis is implemented, and virtually all standard Unix programs operate without change or recompilation. This system is the first, and currently only, existing and functioning kernel based operating system. It conclusively demonstrates the feasibility of this approach. Over the next months, the remaining software development tasks associated with the prototype will be completed and documentation will be developed. A general architecture paper is already available, cited as reference 9.

The UCLA kernel has 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 late last year. Complete abstract specifications were completed early in 1978, and the abstract to concrete correspondence is now well underway. The programs being verified compose the largest practical verification effort in the country, and it is uncovering a great deal about how verification of large systems must be done, as well as the nature of the tools which are required. As this research has progressed, a cooperative arrangement with USC/Information Sciences Institute has developed, since their XIVUS interactive ve-

rification system is being employed in the proofs. The general approach to the verification of large systems is described in the technical report listed as reference 3.

from p.1

The body of this report provides more detail in several areas. Specifically, three reports are included that reflect some of the more significant results in each of the areas of distributed systems, network security and the secure operating system prototype.

## References Published During the Reporting Period

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.

# Encryption Protocols, Public Key Algorithms and Digital Signatures in Computer Networks*

by

Gerald J. Popek and Charles S. Kline
University of California at Los Angeles

## Abstract

The general problem of secure communication in computer networks is considered, especially issues related to integration of encryption protocols, the relationship between public key and conventional encryption algorithms, and digital signatures. The conclusions reached in these areas are as follows.

A) A crucial problem in integrating encryption into networks is minimization of the mechanism which must be trusted. A general protocol is presented which appears to accomplish this goal and is suitable for either public key based or conventional encryption algorithms.

B) Public key and conventional encryption algorithms are functionally equivalent, in the sense that neither present any advantages over the other, either in the way they are used, the functions they provide, or in the amount of mechanism that must be trusted in their support.

C) Both public key and conventional encryption approaches to digital signatures depend critically on secure authentication for their suitability, in ways not generally recognized. They appear equivalent with respect to safety.

D) Neither the signature method outlined by Rabin nor the usual public key based protocols appears satisfactory. A more suitable network digital signature method is described.

## 1. Introduction

There has been considerable interest recently in the development of encryption methods for computer networks. Activity falls into two major but related areas: the development of strong encryption algorithms, and the design of the rules or protocols by which an algorithm is actually used in an operating network. As an example of the relation between these two areas, public key algorithms have been suggested as a superior solution to key distribution and digital signatures; issues which, it is

---

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

claimed, would otherwise require additional protocols. Here we concentrate on the protocol problems. We examine protocol questions which arise at various levels of a system, from the low, detailed level at which the various operating systems in a network communicate, to the higher, user visible level involving such services as digital mail. As a result a rather unique perspective is provided, and we are led to some fairly surprising conclusions.

The paper is written basically in a bottom up fashion. The first section considers questions of how encryption "channels" interact with network software. The next section outlines a basic protocol for the use of encryption in a network, independent of the nature of the encryption algorithm (public key, conventional, etc). These two sections show how it is possible to build a secure network base, on top of which many extensions are directly possible. At that point attention turns to some of the higher level, user visible issues, such as public key algorithms and digital signatures. It is argued that none of the currently proposed signature methods is satisfactory. We propose an alternative which we believe satisfies the necessary requirements. It is based on the existence of the secure lower level protocols discussed in the earlier sections. Those readers willing to accept the existence of secure lower level network protocols may wish to skip to section six, where the discussion of public keys and digital signatures can be found.

## 2. Levels of Integration

Encryption forms the basis for solutions to computer network security problems. Basically, a single communications channel can be multiplexed into a large number of separately protected, secure communication channels by assigning a separate encryption key pair for each logical communication channel. When a user requests the establishment of a new communication, protection policy checks can be performed, and, if successful, a key can be distributed to each end of the communication channel.

Several key distribution methods have been studied.[Popek 78b] One method utilizes a key distribution center which receives requests for communications, and distributes keys accordingly. The keys are transmitted using previously arranged secret keys which change only rarely. Other methods allow distributed key management, with several, or even all, sites participating in key distribution. Recently, public key encryption algorithms [Rivest 77a] have become available. Originally, such algorithms were thought to simplify the key distribution problem, but recent research suggests that no savings result.[Needham 77] This issue is discussed at length in section six.

One problem which must be resolved in designing a secure network encryption mechanism, regardless of the nature of the encryption algorithm or the key distribution method, is the level of integration of the encryption facility. There are many

possible choices for the endpoints of the encryption channel in a computer network, each with its 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 computer systems, referred to as hosts, 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 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, too. This view 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-end encryption.

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

In general, as one considers higher and higher levels in most systems, 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, at a higher level there often are over a thousand processes concurrently operating, each one separately protected and controlled. The number of terminals and users is of course also high. This numerical increase means that the number of secure channels - that is the number of separately distributed matched key pairs required - 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 may result, there can be significant reduction in the amount of software whose correct functioning must be assured for the protection of the communication channel. This issue is very important and must be carefully considered. It arises in the following way. When the lowest level 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, from host to host for example, then errors in the switches are of no consequence. However, operating system failures are still serious, since 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 quite 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 section four.

Another small but nontrivial 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 functions as packing, character erasures, transmission on end of line or control character detect, etc. If the data is encrypted before it reaches the front end, then these functions cannot be performed. That is, any processing of data flowing through the channel must be done above the level at which encryption takes place.

## 3. 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. It is well recognized that choice of protocol has dramatic impacts on the utility, flexibility and bandwidth provided by the network. Since encryption facilities essentially provide a potentially large set of logical channels, the protocols by which the operation of those channels is managed also can have 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 inadvertantly or intentionally leaked by the same means?
3. What facilities are provided for error recovery and resynchronization of the protocol?
4. How is flow control performed?
5. How are channels closed?
6. How do the encryption protocols interact with the rest of the network protocols?
7. 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. Clearly the more software is involved the more one must be concerned about the safety of

the overall network. The performance resulting from use of the protocol must compare favorably with the attainable performance of the network using other suitable protocols without encryption. Lastly, one would prefer a general protocol which could also be added to existing networks, disturbing the transmission mechanisms already in place as little as possible. Each of these issues must be settled independent of the level of integration of encryption which is selected, the method of key distribution, or the nature of the encryption algorithm employed.

To illustrate the ways in which these considerations interact, in the next section we outline a complete protocol. The case considered employs an end to end architecture in a way that can be added to an existing network.

## 4. Network Encryption Protocol Case Study: Process-Process Encryption

We outline here a general encryption protocol that operates at the relatively high level of process to process communication. A major goal is the minimization of the software on which the security of the system depends. Network communication protocols often involve fairly large and complex parts of the operating system, sometimes the primary source of complexity and amount of code. This fact results from the variety of tasks which the network protocol must perform, such as connection establishment, flow control, error detection and correction. Thus, this design attempts to eliminate as much as possible the necessity of trusting that software for secure operation.

The design presented here utilizes process-process encryption. In process-process encryption, encoding is performed as data moves from the source process to the system's network software. This approach minimizes the points where data exists in cleartext form, and thus the mechanism which needs to be trusted. While a higher level choice could be made, for example allowing the processes to perform their own encryption within themselves, such a choice does not assure that all data sent over the network is encrypted. Thus, process-process encryption seems to be the highest safe choice. The details of the protocol are applicable either to public key based or conventional algorithms. Any of the key distribution methods discussed in [Popek 78b] can be supported.

It is assumed that the reader is familiar with the ideas of operating system security kernels.[Popek 78c] Briefly, security kernel based systems attempt to isolate the security relevant parts of the system and place them in a nucleus, running on the bare hardware. In that way, the secure operation of the system depends only on that software. By careful design and implementation of a security kernel, it is possible to formally verify the security properties of the system.[Popek 78a]

## 4.1 Overview

In this protoeol, when a user attempts to send data, a system enerypt function enerypts that data and passes it to the network management software, whieh is logieally part of the local operating system. The network software then attaehes headers or other information required by the network protoeols and sends the data to the eommunieations facility. Upon reeeption by the remote network software, the headers and other protoeol information are removed from the data and the data is passed, via a system deerypt funetion, to the appropriate user proeess.

Initial establishment of the eommunieation ehannel is also provided in a seeure way. When a user proeess attempts to establish eommunieation, the local network software is informed by the system. The network software then eommunieates with the network software at the remote site. When the two network software paekages have arranged for the new eommunieation, the system at eaeh site is informed. At this point in time, the system software attempts to obtain encryption keys for this communieation. This key distribution is aeeomplished either with local key management software, or via a key distribution eenter. If a eonventional eneryption algorithm was employed, then new keys would be ehosen and distributed. If a publie key eneryption algorithm was utilized, then the publie key of the reeipient and the private key of the sender would be retrieved.

In the publie key ease, an additional authentieation sequenee is required, sinee the publie keys may have been used before. This authentieation sequenee effeetively establishes a sequenee number to be ineluded in eaeh message to guarantee that previous messages ean not be reeorded by an imposter and replayed. The authentieation sequenee is not required in the eonventional eneryption ease sinee the new keys effeetively form an authentieation and prevent any prior messages from being useful.

After the keys have been chosen and distributed (using a previously established secure key distribution channel), the user proeesses are given eapabilities to send and reeeive data. The operating system ealls employed should automatieally enerypt and decrypt the data with the appropriate keys. Thus, the communication channel is established.

The above design allows existing network protoeols in many eases to be largely left undisturbed, and preserves much existing network software. If desired, user proeesses ean be blocked, in a reliable way, from eommunieating with any other user proeesses anywhere in the network unless the protection poliey involved in setting up the keys permits it. Eaeh user's eommunication is proteeted from every other user's eommunieation. Perhaps most important, the amount of trusted meehanism required in the system nueleus, as we shall see, is quite limited.

## 4.2 The Encryption Conneetion Protocol

Thc details of seeure communieation establishment, briefly described above, are now presented in morc detail. To outline this procedure, we first view the operation from the vantage point of the operating systcm nucleus, or kernel, and then see how host network protoeol software operates making use of the kernel faeilities. For brevity, in this diseussion, a logical communieation ehannel between two processes will be known as a conneetion. The host network software will be referred to as thc network protocol manager (NPM). In general purpose networks, the role of the NPM is quite sophisticated and requires eonsiderable code to implement the necessary protocols, an important reason not to have seeurity depend on the NPM.

In the discussion below, it will be understood that a pair of matching eneryption keys, one held by each of thc two hosts involved, defincs a seeure, onc way (simplex) channel. A bidirectional (duplex) channel between two hosts therefore employs two pairs of keys.[1] Each kernel of each host in normal operational modc has a securc full duplex ehannel established with eaeh other kernel in the network. How these ehannels are established concerns the method by whieh hosts are initializcd, and is diseussed later. The kernel-kernel ehannels are used for exchanging keys that will be used for other channels between the two hosts and for kernel-kernel eontrol messages.[2] The need for thcse will beeome apparent as the protoeol is outlined. If it is desired, the protocols can be trivially altered to keep the cleartext form of kcys only within the encryption units of the hosts. For simplieity of explanation, that requirement is not used here.

A connection will get cstablished in the following way. When hosts arc initialized, their NPMs will establish connections through a procedurc analogous to the one we outline here, and described in more detail later. Then, when a user process wishes to eonneet to a foreign site, the process executes an "establish connection" system eall which informs the NPM of the request. The NPM exchanges messages with thc foreign NPM using their already existing ehannel. This exchange will include any host-host protocol for establishing communications in the network. Presumably the NPMs eventually agrec that a connection has been established. At that point the user processes are still unable to communicate, since so far as the kernel is concerned, nothing has been done. The eontent of NPM exchanges is invisible to the kernel. Rather, at this point, the NPMs must ask thc kernel to establish the channel for the proeesses. This action is performed with kernel funetion ealls. Those calls grant

---

[1] The same key could be used for both directions in conventional encryption, but for conceptual clarity here it is not.

[2] In a centralized key distribution version, these kernel-kernel secure channels would be replaced by kernel-key distribution eenter seeure channels.

capabilities to the user proeess so that subsequent requests can be made directly by the process.

In order to explain in more dctail, the following four prototype kernel ealls arc described. The first two are involved in setting up the eneryption ehannel, and presumably would be issued only by the NPMs. The second two arc the means by which user processes send and rceeive data over the eonnection.

GID(forcign-host, conneetion-id, proeess-id, statc) Give-id. This call supplies to the kernel an id whieh thc ealler would like to be used as the name of a channel to be established. The kernel checks it for uniqucness beforc aceepting it, and also makes relevant protection checks. If state = "init", the kernel chooses the encryption key to be associated with the id (or queries key controller for key). The entry <connection-id, key, process-id, state> is made in the kernel Key Table. Using its seeurc channel, the kernel sends <conneetion-id, key, policy-info> to the foreign host. The policy-info can be anything, but in the military case, it should be the security level of the local process idcntified by proeess-id. In a eommercial case it might be the organization by whieh the user was employed. It might also be a network-widc global name of the user associated with the proeess. If statc = "eomplete", thcn there should already be an entry in the Key Table (caused by the other host having exccuted a GID) so a check for match is made bcfore sending out the kernel-kernel message and a key is not included. The NPM process is notified when an id is received from a foreign kernel.

CID(connection-id) Close id. The NPM and the appropriate proeess at the local site are both notified that the eall has been issued. The corresponding entry in the Key Table is deleted. Over the secure kernel-kernel channel, a message is sent telling the other kernel to delete its corrcsponding Key Table entry. This call should be executable only by NPMs or by the process whose Key Table entry indicates that it is the process associated with this id, to block potential denial of service problcms.

Encrypt(conneetion-id, data) Encrypt data and buffer for NPM. This call adds integrity information, such as sequence numbers, to the data, encrypts the data using the key corresponding to the supplied id (fails unless the process-id associated with the connection-id matches that of the callcr) and places the data in an internal buffer. Thc NPM is informed of the awaiting data.

Decrypt(connection-id, user-buffer) Decrypt data. This call decrypts the data from the system buffer belonging to the connection-id supplied using the appropriate key. The data is moved into the user's buffer. The call fails unless the process-id stored in thc Key Table matehes the caller and any data integrity ehecks succeed (such as sequenee numbers).

An important new kernel table is the Key Table.[1] It contains some number of entries, each of which have the following information:

<foreign-host, connection-id, key, sequence-no, local-process-id>

There is one additional kernel entry point besides the calls listed above, namely the one caused by control messages from the foreign kernel. There are two types of such messages: one corresponding to the foreign GID call and the other corresponding to a foreign CID. The first makes an incomplete entry in the receiving kernel's Key Table, and the second deletes the appropriate entry.

The following sequence of steps illustrates how a connection would be established using the encryption connection protocol. The host processors involved are numbered 1 and 2. Process A at host 1 wishes to connect to process B at host 2.

1. Process A executes an establish connection call which informs NPM@1, saying "conn from A to B@2". This message can be sent locally in the clear. If confinement is important, other methods can be employed to limit the bandwidth between A and the NPM.
2. NPM@1 sends control messages to NPM@2 including whatever Host-Host protocol required.[2]
3. NPM@2 receives an indication of message arrival, does an I/O call to retrieve it, examines header, determines that it is recipient and processes the message.
4. NPM@2 initiates step 2 at site 2, leading to step 3 being executed at site 1 in response. This exchange continues until NPM@1 and NPM@2 open the connection, having established whatever internal local name mappings are required.
5. NPM@1 executes GID(connection-id, process-id,"init"), where connection-id is an agreed upon connection id between the two NPMs, and process-id is the local name of the process that requested the connection.
6. In executing the GID, 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, who makes corresponding entry in its table and interrupts NPM@2, giving it connection-id.
7. NPM@2 issues corresponding GID(connection-id, process-id', "complete") where connection-id is the same and process-id' is the one local to host 2. This call interrupts process-id', and eventually causes the appropriate entry to be made in the kernel table at host 1. The making of that entry interrupts NPM@1 and process-id@1.

---

[1] In some hardware encryption implementations, the keys are kept internal to the hardware unit. In that case, the key entry in the Key Table can merely be an index into the encryption unit's key table.

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

8. Process-id and process-id' can now use the channel by issuing succeeding Encrypt and Decrypt calls.

There are a number of places in the mechanisms just described where failure can oecur. If the network software in either of the hosts fails or decides not to open the eonnection, no kernel ealls are involved, and standard protoeols operate. A GID may fail because the id supplied was already in use, a protection poliey eheek was not suceessful or because the kernel table was full. The ealler is notified. He may try again. In the ease of failure of a GID, it may be neeessary for the kernel to execute most of the actions of CID to avoid race conditions that can result from other methods of indicating failure to the foreign site.

## 4.3 Diseussion

The encryption mechanism just outlined eontains no error correction facilities. If messages are lost, or sequence numbers are out of order or duplieated, the kernel merely notifies the user and network software of the error and renders the channel unusable. This action is taken on all channels, including the kernel-kernel channels. For every easc but the last, CIDs must be issued and a new ehannel created via GIDs. In the last case, the procedures for bringing up the network must be used.

This simple minded view is aeceptable in part because the expected error rate on most networks is quite low. Otherwise, it would be too expensive to reestablish the channel for each error. However, it should be noted that any higher level protocol errors are still handled by that protocol software, so that most failures can be managed by the NPM without affecting the encryption channel. On highly error prone channels, additional protocol at the encryption level may still be necessary. See Kent [Kent 76] for a diseussion of resynehronization of the sequencing supported by the encryption channel.

From the protection viewpoint, one can consider the colleetion of NPMs aeross the network as forming a single (distributed) domain. They may exchange information freely among them. No user proeess can send or receive data directly to or from an NPM, except via narrow bandwidth channels through which control information is sent to the NPM 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 NPMs, and having the kernel post, as much as possible, status reports directly to the proeesses involved. The ehannel bandwidth eannot be zero, however.

## 4.4 System Initialization Proeedures

The task of bringing up the network software is eomposed of two important parts. First, it is neeessary to establish keys for the secure kernel-kernel channels and the NPM-NPM ehannels.

Next, the NPM can initialize itself and its communications with other NPMs. 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 NPM-NPM 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 NPM-NPM 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 NPM-NPM communications then proceeds as it would if encryption were not present. In most networks, some form of host-host reset command would be sent (encrypted with the proper NPM-NPM key). Once this NPM-NPM initialization is complete, the kernel-kernel connections could be established by the NPM. 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 NPM-NPM 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, NPMs probably should only establish channels with other NPMs when a user wants to connect to that particular foreign site, and perhaps close the NPM-NPM channel after all user channels are closed.

## 4.5 Symmetry

The case study just presented 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

---

[1] In a centralized key distribution version, the only keys which would be needed would be those for the key distributor NPM-host NPM channel and for the key distributor kernel-host kernel channel. In a distributed key management system, keys would be needed for each key manager.

traffic.) Schroeder and Needham have sketched protocols that are similar in spirit to those presented here to deal with such cases.

## 5. Datagrams

The case of electronic mail illustrates an important variation to the protocols presented earlier. Assume that a user at one site wishes to send mail to a user at another site.

Using conventional encryption algorithms, the first user would request a connection to the second user, and a new key would be chosen and distributed by the key controller for use in the communication. That key is sent using the secret keys of the two users.

However, since the second user may not be signed on at the time, a daemon process 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 need to access the cleartext form of the mail, for that would require the mail receiver mechanism to be trusted. This feat 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 that way, the daemon only needs the ability to append to a user's mailbox file.

In order for the user to know the new key used for this mail, however, the key distribution algorithm used earlier must be modified. Rather than sending the key for this connection to both the sender and the receiver, the key controller sends the key twice to the sender, one copy encrypted with the sender's secret key and one copy encrypted with the receiver's. The sender can prepend the copy of the key encrypted in the receiver's secret key to the mail before transmission. When the recipient signs on, his own mail program will examine the mailbox file, find the key message, decrypt it using his secret key, and then use the new key to decrypt the remaining text.

In the case of public key encryption algorithms, the mail problem is somewhat simplified since the recipient knows what key to use in decryption (his secret key). However, authentication is not possible since the recipient is not present when the message is received. Thus, it may be a replay of a previously sent message. This problem can be prevented in the conventional encryption algorithm case via various protocols with the key managers, for example, by timestamping the mail and having the recipient keep track of recently used mail keys.

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 the next section.

## 6. Public Key Algorithms and Digital Signatures

The development of public key based encryption was greeted by a great deal of interest, since the method appears to present considerable advantages over conventional encryption methods, especially with respect to key distribution and digital mail signatures.

However, on closer examination, it seems that public key algorithms possess no particular advantages over conventional algorithms. The reasons for this conclusion are readily seen and are outlined below.

### 6.1 Key Distribution

Let us examine each of the advantages claimed for public key algorithms. The first is key distribution. Simply put, public key advocates argue that an automated "telephone book" of public keys can generally be made available, and therefore whenever user x wishes to communicate with user y, x merely must look up y's public key in the book, encrypt the message with that key, and send it to y.[Diffie 76] Therefore there is no key distribution problem at all. Further, no central authority is required initially to set up the channel between x and y.

Needham and Schroeder point out 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.[Needham 77] Their argument may be summarized as follows. 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 there is no security. Furthermore, maintenance of the (by necessity machine supported) book is non trivial because keys will change; either because of the natural desire to replace a key pair 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.

Needham and Schroeder also exhibit protocols to provide the desired properties for public key systems, and show that there are equivalent protocols for conventional algorithms. The protocols are equivalent both in terms of numbers of messages required as well as in the mechanisms which must be trusted. The only observable difference is that the central authority in the conventional case, in addition to being trusted, must also keep its collection of (conventional) keys secret. Based on the work at UCLA on secure operating systems, it appears that the task of constructing a secure central authority is no harder than building the correct one needed for public key systems.

## 6.2 Digital Signatures

The seeond area in which publie kcy mcthods are often thought to be superior to conventional ones is digital message signatures. The method, assuming a suitablc publie key algorithm, is for the sender to eneodc the mail by "decrypting" it with his private key and then send it. The receivcr decodes thc message by "encrypting" with the sender's public key. The usual view is that this procedure does not require a eentral authority, exeept to adjudieate an authorship ehallenge. However, two points should be notcd. First, a eentral authority is needed by the recipient for aid in deciphering thc first messagc received from any given author (to get the eorresponding public key, as above). Second, the central authority must keep all old values of public keys in a reliable way to properly adjudicate conflicts over old signatures (ccnsider the relevant lifetime of a signature on a real estate deed for example).[Needham 77]

Further, and more serious, the unadorned public key signature protocol just deseribed has an important flaw. The author of signcd messages ean effectively disavow and repudiate his signatures at any timc, merely by eausing his secrct key to be made public, or "compromiscd". When sueh an event occurs, either by accident or intention, all mcssages 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 of the messages sent earlier 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 history of protection of the privatc key. Further, the ability to remove the proteetion resides in precisely the individual (the author) who should not hold that right. That is, one important purposc of a signature is to indicate responsibility for the eontent of the aeeompanying message in a way that cannot be later disavowed.

Some people may argue that this concern is overly eonservative; that existing signature methods are not very reliable, that individuals have considerable incentive not to repudiate their signatures, and so one is justified in constructing a flawed solution. However, in our view this characteristic is clearly unsatisfactory, especially if it is possible to devise suitable digital signature methods which do not suffer from this problem.

The situation with respect to signatures using eonventional algorithms initially appears slightly bctter. Rabin [Rabin 78] proposes elscwhere in this volume a method of digital signatures based on any strong eonventional algorithm. Like public kcy methods it too rcquires 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.

## 6.3 A Reliable Digital Signature Method

A simple, obvious solution is to interpose some trusted interpretive layer between the author and his signature keys, whatever their form. For example, suppose the list of keys in Rabin's algorithm were not known to the author, but instead were contained in a secure Unit (hardware or software). Whenever the author wished to send a signed message, he merely submitted the message to the Unit, which selected the appropriate keys and then used the standard algorithm. Each author has access to such a Unit.

The loading of each Unit requires some examination. In particular, the means which are used to select keys and insert them into each Unit must be correct if mail challenges are to be handled satisfactorily. That is, there must be some trusted Source of keys (and matching "standard message" in the Rabin protocol), and the key list for each author/recipient pair must be deliverable in a correct, secret way to the appropriate Units. We will call the collection of Units and the Source(s), together with their internal communication protocols, a Network Registry (NR). Such an NR appears required to solve the problems raised earlier. Note that some secure communication protocol among the

---

[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.

components of the Network Registry is required. However, it can be very simple; low level link encryption would suffice.

For safety and efficiency, the NR functions presumably should be decomposed and distributed throughout the network. In particular, the failure or compromise of a local NR would then only have local consequences. One can even construct local NR components of the Network Registry in a decentralized way so that compromise of more than one component would be required before a message signature was affected.[1] The NR architecture issue, while important, is to some degree a digression here and so we put it aside.

The Registry concept is quite common in the paper world. A local government's real estate recorder's office is probably the most commonly known example.

## 6.4 Authentication

We now make an important observation. It is still necessary that there exist a guaranteed authentication mechanism by which an individual is authenticated to the NR (presumably directly to the local Unit). Any reasonable comunication 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.

## 6.5 Simplification of the Proposed Signature Architecture: Specialized Digital Signature Protocols Unnecessary

**Once the necessity of a Network Registry is recognized, including a guaranteed authentication mechanism, it appears that simplifications in the mechanisms required for digital signatures can be made that seem to remove the need for specialized digital signature protocols. Instead, any of a collection of simple methods will suffice.**

In particular, in order for the Network Registry to operate satisfactorily (including performing user authentication), it clearly must be distributed, and clearly must be able to communcicate securely internally among the distributed components. Given that such facilities exist, then the following is an example of a simple implementation of digital signatures which does not require a specialized protocol or encryption algorithm:

1. The author authenticates with a local Network Registry

---

[1] See section 6.6.

component, 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. A 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.

This simple protocol involves little additional mechanism beyond that which was needed by the Network Registry anyway. It does require that the Network Registry be involved in every message signature and validation. However, recall that all of the unadorned signature methods reviewed earlier require involvement of some form of a Network Registry for at least the first message between any two parties. Public key protocols must check the "telephone book", and Rabin's method requires either a contract or a Network Registry. Furthermore, when one adds a more complete Network Registry on top of those other signature methods to correct their repudiation problem, all methods involve the NR for each message. Note that this protocol also does not require the NR to maintain any significant storage for signature blocks.

## 6.6 Performance and Safety

Certain elementary precautions should be taken in the design of the Network Registry to avoid unnecessary internal message exchanges and to assure safety of the keys used to encrypt the signature blocks. Performance enhancements presumably would involve distributing the signature block calculation. Safety enhancements could include the use of different keys at each distributed site, replicating sites, and employing a signature block computation which requires the cooperation of multiple sites. Each of these facilities is straightforward to build and so they are not discussed further here.

From the preceding discussion, we conclude that the digital signature algorithms proposed heretofore are unsatisfactory, and the improvements required to correct their inadequacies make the use of a specialized digital signature algorithm unnecessary.

We note here that the safety of signatures in this proposal also depends on the future history of protection of keys as before, in this case those held by the Network Registry. However, there are several crucial differences between this case and previous proposals. First, the authors of messages do not retain the ability to repudiate signatures at will. Second, the

Network Registry can be structured so that failure or compromise of several of the components is necessary before signature validity is lost. In the previous methods, a single failure could lead to compromise.

## 7. Conclusions

We draw a number of specific conclusions, as well as more general perspectives from the preceding discussions. The specifics are as follows. First, public key encryption systems, viewed in the context of the network protocols by which they must be used, do not seem to provide any significant advantages over conventional encryption algorithms. Each important function that has been recognized can be performed at least as easily by conventional methods with, it appears, no more supporting mechanism. Therefore, if strong conventional algorithms are easier to develop, as has been speculated [Rivest 77b], research would be better devoted to that area rather than public key systems.

Second, it seems that the digital signature methods which have been proposed, both public key and conventional algorithm based, do not adequately protect recipients of signed documents from repudiation of signatures by the author revealing the secret key(s) employed. The difficulty appears intrinsic to the approaches being taken. An alternative is available which overcomes this problem however, that involves a small amount of trusted software.

Third, the necessary underlying mechanism required to support improved digital signature methods, as well as other user visible secure network communcation protocols, is relatively well understood, and an example is presented in this paper. The example takes account of the important requirement that the amount of trusted mechanism involved be minimized for the sake of safety.

In more global terms, this discussion of network security has been intended to illustrate the current state of the art. It suggests the following general perspectives.

If one's view of security of data in networks is basically a common carrier philosophy, 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 one important assumption that is not universally valid. Either there exist secure operating systems to support the individual processes and the required encryption protocol facilities, or each machine operates as a single protection domain. A secure implementation of a Key Distribution Center or Registry is necessary in any case. Fortunately, reasonably secure operating systems are well on

their way, so that this intrinsie dependency of network sceurity on an appropriate operating system base should not seriously delay eommon carrier security.

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 maehine for the user, in whieh no explicit awareness of the network is rcquired. The underlying facility is trusted to seeurely move data from sitc to site as necessary to support whatever data types and operations that are relevant to the user. Thc facility operates securely and with integrity in the faee of unplanned crashes of any nodcs in the network. Synchronization of operations on user meaningful objects (sueh as Withdrawal on CheekingAccount) is reliably maintained. If one takes sueh a high level vicw of the goal of network security, then the simple common carrier solutions respond only to part of the network security problem and more work remains.

## 8. Bibliography

[Diffie 76] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transaetions on Information Theory, November 1976, 644-654.

[Kent 76] Kent, S., Enerypton-based Proteetion Protoeols for Interaetive User-Computer Communication, Laboratory for Computer Science, MIT, TR 162, 1976.

[Needham 77] Needham, R. and M. Sehroeder, Security and Authentieation in Large Networks of Computers, Xerox Palo Alto Researeh Center Teehnieal Report, September 1977.

[Popek 78a] Popek, G. J. and D. Farber, "A Model for Verifieation of Data Sccurity in Operating Systems", Communications of the ACM, (to appear).

[Popek 78b] Popek, G. J. and C. S. Kline, "Design Issues for Seeure Computer Networks", in Operating Systems, An Advanced Course, R. Bayer, R. M. Graham, G. Seegmuller, ed., Springer-Verlag, 1978

[Popek 78c] Popek, G. J. and C. S. Kline, "Issues in Kernel Design", Proceedings of the National Computer Conferenee, AFIPS Press, 1978

[Rabin 77] Rabin, M., "Digital Signaturcs Using Conventional Encryption Algorithms", Proeeedings of the Conferenee on Foundations of Seeure Computing, Atlanta Gcorgia, October 3-5, 1977, Academic Press (to appear).

[Rivest 77a] Rivest, R. L., Shamir, A., and L. Adlcman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, MIT Laboratory for Computer Seienee Teehnical Memo LCS/TM82 Cambridge, Mass. 02139 April 4, 1977 (Revised

August 31, 1977)

[Rivest 77b] Rivest, R., private communications, 1977.

# A LOCKING PROTOCOL FOR RESOURCE COORDINATION IN DISTRIBUTED DATABASES*

Daniel A. Menasce†, Gerald J. Popek and Richard R. Muntz

Computer Science Department
University of California
Los Angeles, California 90024

ABSTRACT: A locking protocol to coordinate access to a distributed database and to maintain system consistency throughout normal and abnormal conditions is presented in this paper. The protocol is robust in the face of failures of any participating site and in the face of network partitioning. The proposed protocol supports the integration of virtually any locking discipline including predicate locking methods. A cost and delay analysis of the protocol as well as a proof of its correctness is included in this work. The paper concludes with a proposal for an extension aimed at optimizing operation of the protocol to adapt to highly skewed distributions of activity.

KEYWORDS AND PHRASES: concurrency, crash recovery, distributed databases, locking protocol, consistency.

## Introduction

This paper is concerned with issues of resource coordination in distributed systems, and the maintenance of system consistency throughout normal and abnormal conditions. A database is said to be in a consistent state if all the data items satisfy a set of established assertions or consistency constraints. A database subject to multiple access requires that accesses to it be properly coordinated in order to preserve consistency. Coordination of resources in a distributed environment exhibits additional complexity over resource coordination in centralized environments due to:

1. possibility of crashes of participating sites and or communication links. Occurrence of such failures can render the database inconsistent if not appropriately handled by the coordination algorithm.

2. network partitioning: in general, it is not possible to distinguish between messages which could not be delivered due to a crash of the recipient site and undelivered messages due to network partitioning. Therefore, network partitioning in the more general sense considered here is not simply a matter of proper network topology design. It turns out that detection of network partitioning can only occur at network reconnection time.

3. inherent communication delay: the time to get a message through a computer communication network may be arbitrarily long, although finite. Therefore any proposed solution should operate correctly regardless of the delay experienced by any message, and in general should be efficient.

A protocol to coordinate concurrent access to a distributed database using locking is presented in this paper. The algorithm has as its core a centralized locking protocol with distributed recovery procedures. A centralized controller with local appendages at each site coordinates all resource control, with requests initiated by application programs at any site. Recovery is broken down into three disjoint mechanisms; for single node recovery, merge of partitions and reconstruction of the centralized controller and tables.

Among the properties of the proposed protocol we have:

a. robustness in the face of crashes of any participating site, as well as communication failures, is provided. The protocol can recover from any number of failures which occur either during normal operation or during any of the three recovery processes.

---

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

† Partially supported by the Conselho Nacional de Desenvolvimento Cientifico e Tecnologico, CNPq, Brazil.

b. deadlock prevention and or detection methods can be easily integrated given the centralized control characteristic of the proposed algorithm.

c. straightforward integration of predicate locking methods [1] is permitted. Value dependent lock specification at the logical level is necessary to avoid the problems of "phantom tuples" discussed by Eswaran et al [1]. Other locking disciplines may also be easily supported.

d. continued local operation in the face of network partitioning is supported. The locking algorithm operates, and operates correctly, when the network is partitioned, either intentionally or by failure of communication lines. Each partition is able to continue with work local to it, and operation merges gracefully when the partitions are reconnected.

e. performance of the algorithm does not degrade operations. It is shown in this paper that for many topologies of interest, the delay introduced by the protocol is not a direct function of the size of the network. The communication cost is shown to grow in a relatively slow, linear fashion with the number of sites in the network.

f. the correct operation of the protocol in the face of the failures mentioned before can be proven in a straightforward way.

Several other approaches for synchronization in distributed databases have been suggested in the literature, but none deal satisfactorily with all of these issues.

The Majority Consensus protocol proposed by Thomas[8] requires the sites involved in a transaction to agree by majority vote for it to proceed. Timestamps on data items at each site indicate whether the item is current and therefore whether a transaction based on it can be approved.

This protocol is quite elegant, with attractive behavior in the face of failures, especially for fully replicated databases. Unfortunately, for the cases considered in this paper, it presents several drawbacks. The locking discipline and scheduling of transactions are fixed by the nature of the algorithm itself, limiting flexibility (predicate locking cannot be supported for example). Performance can degrade severely with increasing system load in a thrashing like manner, since several partially complete transactions which conflict lead to multiple resubmission of each.

Synchronization in SDD-1 [4] is handled by several different protocols designed to co-exist with one another. The simpler ones can be used for certain restricted classes of transactions known in advance of system generation. In such cases significant improvements in cost and delay over more general protocols results. Otherwise however, we recommend our protocol since its performance is absolutely better and issues such as robustness and crash recovery, not handled by SDD-1, are considered fully.

A ring structured solution is proposed by Ellis[6]. It uses sequential propagation of synchronization and update messages along a statically determined circular ordering of the nodes. Two round trips are required for each update. This protocol, while in general much slower than the others mentioned above, is quite simple and Ellis has employed formal verification procedures to show its correctness. Unfortunately however, failures and error recovery are not addressed by the protocol.

Other proposed schemes, called primary copy strategies have been suggested in [3], [5] and [7]. Alsberg in [3] introduced some techniques aimed at providing a certain degree of resiliency to the single primary, multiple backup strategies discussed in [5] and [7]. The primary copy scheme is primarily designed to maintain mutual consistency of databases subject to somewhat limited types of update operations, but it does not address explicitly the problem of internal consistency of a distributed system supporting general transactions.

The protocol presented in this paper is described in an intuitive manner in section one, followed by a more detailed description in the two subsequent sections. An algorithmic specification of this locking protocol can be found in [2]. An informal proof of the correctness of the algorithm is presented here. The proof is decomposed into five major parts, one for normal operation, three for the recovery phases, and a last part that shows the parts actually can be proved disjointly.

The paper concludes with a proposal for an extension aimed at optimizing operation of the algorithm to adapt to highly skewed distributions of activity. The extension applies nicely to interconnected computer networks.

## 1 - Centralized Lock Controller Protocol - Intuitive Description

The database we are considering here is distributed among n nodes of a computer network, numbered from 1 to n. We assume that the network protocols are such that a copy of a message is kept by its sender until an acknowledgment for it is received. In other words, there are no lost messages. However, messages may have to be retransmitted many times until they get through the net. An implication of these assumptions is that messages may be delayed by an arbitrary but finite amount of time. We also assume that messages from a source site A are delivered to a destination site B in the same order they were generated. However, we make no assumptions about the order in which messages from two distinct sources are received by a third one. We require that the network routing procedures be such that every pair of nodes can communicate with each other if the necessary physical connection is available.

User interaction with the database is done

through application programs, APs, which communicate with the Data Base Management processes. Of those processes, two are of interest for this locking protocol: the 'centralized lock controller' or simply 'lock controller' and the 'local lock controller'.

As a first approximation assume that there is only one lock controller or LC for the entire network. This process is responsible among other things for examining lock and lock release requests from the APs, and deciding whether they should be granted or not. For this purpose, the LC maintains a table called the LOCK table, which is a set of all the active locks. Each entry in this table is a 3-tuple of the form (H,T,P) where H is a unique host identification, T is a unique transaction identifier within each site and P is a description of the logical portion of the database to be locked as well as the lock mode (e.g., read, write, etc.). In a relational database, the lock specification may for example be a predicate lock as described by Eswaran et al [1].

At every site, except for the one where the LC is located, there is a local lock controller or LLC. Those processes are responsible for maintaining a local copy of the LOCK table. Any LLC may become the lock controller whenever there is a crash in the system which makes the LC unavailable. The recovery process is explained later in detail. Each time a transaction takes an action the local copy of the LOCK table is examined to determine whether the action can be performed or not. Therefore, there are two reasons for keeping a local copy of the LOCK table, namely: resilience to failures and local action checking.

It is convenient at this point to introduce the notion of logical partition or logical component, as opposed to that of a physical component. A physical component is a maximal subnetwork such that every pair of sites in the component can communicate with one another. It can be readily seen that the composition of a physical component is not under the control of the locking protocol, since nodes and communication links fail independently of the protocol operation. Such a lack of control could make the operation of the protocol, in the face of crashes, rather complex. The concept of logical component is introduced to give the protocol independence from unexpected changes in the composition of each physical component. To this end, each LC keeps a list of sites which he thinks are still up, called the up list. A logical component is defined as being the subnetwork indicated by the nodes which are in the up list. This list may lag behind the list of sites which are actually up. Independence from the composition of physical components is thus achieved by controlling the way by which the latter list maps into the former, in a way which is explained later in the paper.

Since one of our stated goals is to allow local operations to continue in face of network partitioning and to allow partitions to merge gracefully, it is necessary for each partition to have its own LC. There is one LC for each logical component.

The operation of the locking protocol under no crash conditions can be intuitively explained as follows. The LC receives lock and lock release requests from the application programs. Each request is sent to all LLCs in the component. The request is stored in a pending list at each LLC site and an acknowledgment is sent back to the LC. After the acknowledgment from all sites in the component is received (excluding those which crashed in the meantime) a confirmation for the request is sent by the LC to all LLCs causing the request to be deleted from the pending list and appended to the LOCK table.

A lock request may be rejected by a LC if it conflicts with other locks in the LOCK table or if the request is not local to the component. We assume that the LC is able to determine for each lock, P, the set, LOC(P), of sites where the data to be locked are stored. Thus, a lock P is said to be local if LOC(P) is contained in the up list for the component. The set LOC(P) can be determined by the LC by checking some catalogs. The organization of those catalogs is not relevant here; see [9] and [10] for discussions of that subject.

Every time that a site or a set of sites drop out of the up list, all the locks which are not local any more are released and all the transactions which had at least one lock released will be aborted or backed up. In this way complete locality of operations is enforced by the CLC protocol.

If the LC crashes or becomes unavailable a recovery mechanism called Logical Component Recovery (LCR) takes place. As soon as an LC-crash is detected by any process engaged in a conversation or exchange of messages with the lock controller, a new process is nominated to be the new LC. There is a globally known circular ordering of the sites from which the nominee is selected. If the nominee is up it accepts the nomination by sending a message which circulates through all the sites in the component. The purpose of this message is also to collect all the requests which have been received by all the sites but which are still in the pending list for at least one site. Those requests will be incorporated into the LOCK table at every site in a subsequent phase of the recovery process. In summary, the LCR mechanism amounts to electing a new LC for the component and bringing all the LOCK tables to the same value before normal operation is resumed. Various race conditions are dealt with by the details of the recovery protocol.

It is the responsibility of each LC to periodically monitor the connection between it and a node not in its up list. If a physical connection between two previously logically disconnected component is detected, a Logical Component Merge (LCM) mechanism is started. LCM is always done pairwise between components and in this process the LC of one of the components plays an active role while the other plays a passive one. The first phase of LCM is composed of an interconnection protocol by which two LCs are logically connected in such a way that one of them is designated active and the other passive. This protocol also enforces the pairwise merge condition and is shown to be deadlock free. After a logical connection has been established both LCs clear all

outstanding requests and reject further ones. In the subsequent phase, the union of the LOCK tables of the two components is made and the new LOCK table is sent to all the sites in both components in the form of a message which circulates through them. This message signals the completion of the merge. The active LC becomes the lock controller for the new logical component.

When a site which was down recovers, it is made active by the Single Node Recovery (SNR) mechanism which basically amounts to the acquisition by that site of a new copy of the LOCK table.

The three recovery mechanisms described above do not interact with each other, as will be shown later. This property is important because it allows us to decompose the correctness proofs into a proof of disjointness and then proofs for each recovery procedure separately.

The recovery mechanisms will be shown to be robust in the face of additional failures. In order to achieve this goal, each mechanism is designed in such a way that a partial execution of any of the recovery algorithms does not destroy any of the properties we want to prove about them.

It is important to emphasize at this point that, since all the lock requests are examined by an LC in each logical component, locks granted by LCs do not conflict with one another. This fact enables us to consider the operation of the algorithm for normal operation and for recovery as if there were only one lock per logical component. The reader is encouraged to keep this in mind as he reads through this paper.

## 2 - Lock and Release Granting Algorithms

This section describes informally the algorithms used to grant new locks and to release existing ones. One would like those algorithms to have the property that a lock is either granted or released if and only if it is known to all the sites. The basic structure of both algorithms can be abstracted in what we call the Assured Communication Protocol (ACP) which exhibits the desired property outlined below.

Let there be a sender S, who wishes to send a message M, originated at an external source ES, to n destinations D1, D2, ..., Dn. Each site i keeps two message buffers: temp_buffer(i) and final_buffer(i). ACP is such that message M will only be in final_buffer(S) iff M is either in temp_buffer(Di) or final_buffer(Di) for all destinations Di. ACP can be described by the following set of rules:

1. S receives a "MESSAGE REQUEST" or MR message from ES and broadcasts an "ACCEPT MESSAGE" or AM message, which contains M, to all Di's, i=1,...,n. The message M is placed in temp_buffer(S).

2. When an AM message is received by a destination Di, the message M is placed in temp_buffer(Di) and a "MESSAGE ACCEPTED" or MA message is sent back to S.

3. When all the MA messages have been received by S, M is moved to final_buffer(S) and removed from temp_buffer(S) and a "CONFIRM MESSAGE" or CM message is broadcast to all destinations.

4. The receipt of a CM message at destination Di causes M to be moved into final_buffer(Di) and removed from temp_buffer(Di).

A variant of this approach with additional acknowledgment messages, called a two-phase commit protocol, is described in [11] and [12].

Several other details are also worth keeping in mind. As mentioned before, each LC keeps a list of the sites in the component which are up. A node i is removed from this list by the LC each time that the underlying network protocols fail to deliver a message to site i (after timeout and retransmission occurred a certain number of times). An up list is also modified by the execution of any of the three recovery mechanisms. A copy of the up list is also kept by each LLC. Every update to the up list by the LC is transmitted to all LLCs in the component. Note that no additional message traffic is generated by those updates since they can "piggyback" on other messages. The reason for keeping local copies of the up list is merely a matter of performance, since the up list determines to some extent the set of nodes which should participate in the LCR or LCM recovery mechanisms, as will be seen later. Also, every time that a change in the up list causes certain locks not to be local any more, all non-local locks are released and the affected transactions aborted.

### 2.1 - Lock Granting Algorithm

Application programs issue lock requests by sending a "LOCK REQUEST" or LR message to the LC. This message contains the lock or 3-tuple which the user would like to be entered in the LOCK table. The LC decides whether the lock can be granted or not. If the requested lock conflicts with other active locks a scheduling decision must be taken by the LC as to whether to preempt any transaction or to make the requester wait. That decision is not the concern of this paper. If there are no conflicts and the lock is local to the component the LC must notify every LLC in its component that a new entry should be appended to their LOCK tables. Actually, instead of inserting the lock directly into the LOCK table, an LLC appends it to a list of pending lock requests, called an L-list. The reason for this is to prevent copies of the LOCK table from becoming inconsistent if the LC crashes.

The basic structure of the Lock Granting and Lock Releasing algorithms is the same as that of the ACP protocol, where AP, LC, LLCi and LOCK table correspond to ES, S, Di and final_buffer in ACP, respectively. Also, the message M in ACP should be considered as a lock request for the Lock Granting algorithm and as a release request for the Lock Releasing one. For the Lock Granting Algorithm, in particular, temp_buffer corresponds to an L-list.

## 2.2 - Lock Releasing Algorithm

A similar procedure is followed when an AP issues a lock release request, by sending to the LC a "RELEASE REQUEST" or RL message. Each site keeps a list of pending release requests or an R-list for the same reasons we introduced the L-list. The R-list corresponds to temp_buffer in the ACP protocol.

## 2.3 - Some Definitions and Proofs

We will show here that, if no crash occurs, the Lock Granting and Lock Releasing algorithms have the property that a lock is only granted or released if all the sites in the component know about the request. In order to make this statement more precise consider the following definitions. Let LT(i), L(i) and R(i) be the LOCK table, L-list and R-list at site i respectively.

DEFINITION 1 (Lock Request Presence): A lock request or a lock is said to be present at site # i, if the lock is either in LT(i) or if it is in L(i).

DEFINITION 2 (Release Request Presence): A lock release request is said to be present at site # i if it is either in R(i) or if is not in LT(i).

The proof for the following two assertions, as well as for all other assertions in this paper, can be found in [2].

ASSERTION 1: If a lock is in LT(i) for some i=1,...,n and in the L-list for at least one site, then this lock is present in every other site of the component.

ASSERTION 2: Let x be a lock and y its associated release request. If x is in LT(i) for at least one site in a logical component but not in all of them and y is in at least one R-list, then y is present in every other site.

Assertions 1 and 2 together lead directly to the following result.

THEOREM 1: Let C be a logical component, LC its lock controller and U the set of sites in C. If no crashes ever occur then a lock request is only granted by the LC after it is present at all the sites in U and a lock is only released if the associated release request is present at every site in U.

## 3 - Crash Recovery

So far we have described the protocol for requesting locks and releasing them, assuming that no crash occurred. Communication links, processors, operating systems and processes are some examples of sources of crashes.

The three already mentioned recovery mechanisms will be presented here. These mechanisms will be proven to be robust with respect to additional failures. To be robust, the protocols must preserve logical component internal and mutual consistency as defined below, if any changes have been made to any permanent information (like LOCK tables, up lists or LC id's) at any node.

DEFINITION 3 (LT-consistency): The set of LOCK tables of a Logical Component is said to be LT-consistent if assertions 1 and 2 hold at any time.

DEFINITION 4 (Logical Component Internal Consistency): A logical component is said to be internally consistent if the set of its LOCK tables is LT-consistent and if there is one and only one LC, whose identity is known to every node in the component.

DEFINITION 5 (Logical Component Mutual Consistency): A set of logical components is said to be mutually consistent if all of them are internally consistent and if there is no lock present at any LOCK table of one of them which conflicts with another such lock of any other component.

Definition 5 covers the previous two, and specifies the property which is required of recovery.

The recovery protocols have been designed so that all crashes which can occur during a recovery phase fall into one of the two disjoint classes, which we call terminal and transparent failures.

A terminal crash causes the entire recovery mechanism to be aborted and restarted. The possible conditions under which terminal crashes occur are shown to leave the protocol in a robust state, as defined above. A transparent crash is defined to be one which does not affect the continued correct operation of the recovery process.

Therefore, if all crashes can be shown to be either terminal or transparent, the recovery protocols are robust. As we will see, for each of the recovery mechanisms, we can identify a point before which the recovery can be considered as not having happened at all and after which it is considered to be successfully carried out. This point is called the 'completion point'. Crashes before the completion point, if they have any effect at all, are shown to be terminal. Crashes after the completion point are shown to be transparent.

The three proposed recovery mechanisms will be shown to occur disjointly in time. In other words, a merge of two logical components only takes place if both are in their normal state or are not recovering from a Logical Component Crash. Also, a site only becomes attached to a logical component if this component is in its normal state. These important properties will allow us to state and prove separate theorems concerning each one of them.

## 3.1 - Logical Component Recovery (LCR)

We will now show how an LLC may become an LC if the LC crashes. A crash of the LC can be detected by any process engaged in a conversation or exchange of messages with it. As an example, an

AP may time-out while waiting for a reply from the LC for a lock or lock release request. In every case, the process which detects a crashed LC is responsible for nominating a new LC. For this purpose, we will assume that the distinct sites or nodes in the underlying network are arranged in a linear order such that node #i precedes node #(i+1) mod n. Let this order be called the nomination order. So, whenever a process detects a failed LC it nominates the next node which is up in the nomination order to the position of LC. This nomination is accomplished by the issue of an "ACCEPT NOMINATION" or AN message by the nominator. If this message is not acknowledged after a certain number of times it has been retransmitted, the nominator assumes that the nominee is down and sends an AN message to the next site in the nomination order. However, it may be the case that the originally nominated node was not down, as assumed by the nominator, but that due to certain conditions in the network its reply was seriously delayed. So, it seems that more than one LC could be nominated in this process! Let us neglect this issue for the moment, while we describe the recovery procedure, and show later how such an undesirable situation can be easily avoided. The nominee is first responsible for checking that the old LC is actually dead (since the nomination may have come from an errant AP). Then the nominee must notify every other site that it has accepted the nomination. Moreover, the nominee must make sure that all the copies of the LOCK table be made equal to the one held by the crashed LC. From now on, we will refer to the crashed LC as the 'old LC' and to the nominee as the 'new LC'.

The process by which the new LC becomes the actual LC can be divided into two phases: a 'notification phase' and a 'LOCK table update phase'.

In the notification phase all the nodes in the component, as indicated by the up list U, are informed of the identity of the new LC. Also, in this phase enough information is gathered in order to appropriately update the LOCK tables in the subsequent phase. The necessary information is described by the sets L and R as defined below.

DEFINITION 6 (set L - set of locks to be added to all LTs):

$$L = \{ x \mid x \text{ is in } L(i) \text{ for some } i \text{ in } U \text{ and } x \text{ is present in all sites in } U \}$$

So, a lock x is in L if it is present at every site but it is in at least one L-list, which implies that all the sites received an "ACCEPT LOCK" message from the old LC, but at least one did not receive a "CONFIRM LOCK" message.

DEFINITION 7 (set R - set of locks to be deleted from all LTs):

$$R = \{ x \mid x \text{ is in } R(i) \text{ for some } i \text{ in } U \text{ and } x \text{ is present in all sites in } U \}$$

So, if a release request is in R, then all the sites in U have already received an "ACCEPT RELEASE" message from the old LC.

The new LC, upon nomination, will issue a message called "NOMINATION ACCEPTED". This message will circulate once through the set of all sites in U (including the site where the new LC runs) in a predetermined order.

In order for the set L to be constructed, two sets, L1 and L2, are formed during the NA cycle. L1 is the set of locks which are present at all sites, while L2 is the set of locks which are in all the LOCK tables. By definition 6, the set L is the difference between L1 and L2.

The set R is also made out of two sets R1 and R2. R1 is the set of lock release requests which are not present in at least one site, and R2 is the set of lock release requests in the R-list of at least one site. The difference R2 - R1 is the set of locks which are present at every site, which by definition 7 is the set R.

Every node, other than the newLC, in the NA cycle receives partially constructed sets L1, L2, R1 and R2, adds its contributions to them and places the new versions of the sets into the NA message which is forwarded to the next node in the cycle. When the NA message returns to the newLC, the sets L and R are completed. Also, the up list U for the new LC will be initialized with the sites which participated in the above described cycle.

After the notification phase is over, the new LC will send a message to every LLC asking them to update their LOCK tables. This message is called an "UPDATE TABLE" or UT message, and it carries within it the sets L and R.

Having updated the LOCK table, each LLC sends a "TABLE UPDATED" message or TU message to the new LC. After receiving a TU from every up site the new LC becomes the actual LC by notifying all the LLCs that they can resume their normal activity. For this purpose the LC broadcasts a "RESUME NORMAL ACTIVITY" or RNA message. The new value for U is the set of sites from which the LC received a TU message. This new value for U is included in the RNA message, thus allowing every node in U to know the composition of the set U.

Let us now describe how we can guarantee, and in effect, prove that only one LC will emerge from the notification process. Recall that the nominator will nominate the first up node in the nomination sequence. Let us make the following definition:

DEFINITION 8 (trial sequence, T[j,k]): A trial sequence, T[j,k], is the sequence i[1], i[2], ..., i[k-1] of site numbers for which an "ACCEPT NOMINATION" message has been unsuccessfully sent by a nominator j, before j sent an AN message to site #k.

For every AN message sent from site #j to site #k we include the sequence T[j,k] as part of it. This sequence will also be included as part of the "NOMINATION ACCEPTED" message which circulates through the set of sites. The purpose of this is to allow any site to resolve any conflict that can arise due to the race conditions discussed earlier in the paper. Namely, it is possible that more than one LC was nominated and consequently more

than one NA message (from distinct sources) would be circulating. Conflicts are resolved by giving preference to the last LC to be nominated. NA messages originated by other nominated LCs are killed when they are detected to belong to the improper LC.

In many instances, in the CLC protocol, we require a certain message to circulate through a set of nodes, as it is the case of the NA message. Let us call such messages 'circular messages'. They always have a source or generator who is responsible for sending it through a cycle. The underlying network protocols assure us that messages will not get lost while going from one site to another by the use of time-out and retransmission schemes. However, a circular message can still be lost if a node in the cycle crashes after receiving it but before being able to forward it. The loss of a circular message can be prevented by having each node in the cycle send to the circular message generator a copy of it, but only after it was forwarded to the next node in the sequence. Now, the source is able to detect a cycle interruption and it can appropriately resume it by sending the last copy of the message to the appropriate site. This source acknowledgment scheme at the CLC protocol level will be assumed to exist whenever a circular message is necessary.

It should be noted that if an application program issues a lock or release request and the LC fails before the request is present at every site, the request will never appear in the local LOCK table even after the LCR is completed. Therefore, APs should timeout for requests and resubmit them.

### 3.2 - Proofs About LCR

We would like to prove now that the notification phase ends with one and only one LC having been successfully nominated, and that all sites know the correct new LC identification. As a first step we state assertions 3 and 4 which are concerned with the behavior of LCR given that no additional crashes occur.

ASSERTION 3: Given that no additional crashes occur during LCR, there will be one and only one LC whose identification is known to all sites in the component at the end of the notification phase.

The proof for this assertion is based on the operation of the trial sequence mechanism described above.

Next, let a globally accepted lock (release) request be one which is in all L-lists (R-lists) of a logical component.

ASSERTION 4: Given that no additional crash occurs, the following is true at the end of the LCR mechanism. All the copies of the LOCK table for a logical component are identical to the value that the LOCK table of the crashed LC would have if all the globally accepted requests were allowed to complete before the crash of the LC.

The proof for this assertion considers a snapshot of all LOCK tables when a crash occurs. It is first assumed that there are no globally accepted requests. In this case, the union of the LOCK table of the crashed LC, LT(oldLC), with the LOCK table of a given site i, LT(i), is considered. It can be shown that all the locks in LT(oldLC) but not in LT(i) will be included in LT(i) by LCR. Also, all the locks in LT(i) but not in LT(oldLC) are removed from LT(i) by LCR. Finally, all the locks in LT(i) and LT(oldLC) are not affected by the LCR mechanism. If there are globally accepted requests they will be included in the sets L and R by definition of these sets. Therefore, the LOCK table of all the sites in the component will be updated in exactly the same way that LT(oldLC) would have been if all globally accepted requests had completed. Given these assertions we prove the robustness of the LCR mechanism.

THEOREM 2: The Logical Component Recovery (LCR) algorithm is robust.

Proof: The completion point for this algorithm occurs when the LC has already sent all the RNA messages. The only terminal crash is a newLC failure before this point. This crash when detected will cause another LC to be nominated and the LCR mechanism to be restarted. This crash can occur at three different points:

i) before any LOCK table has been updated.
ii) after some but not all LOCK tables have been updated.
iii) after all LOCK tables have been updated.

In case i) it is clear that the partially executed LCR has no effect at all. In case iii) all LOCK tables will be identical, therefore internal consistency for the component in question is trivially satisfied. Case ii) requires us to show that the set of LOCK tables of a component is LT-consistent. We enunciate and prove this statement as the following lemma.

LEMMA 1: Given a logical component where the set of LOCK tables is LT-consistent, then the update of the LOCK table as indicated by the sets L and R in some but not all of the nodes of the component preserves LT-consistency.

Proof: Let i be a site for which the LOCK table has been updated. The LOCK table is updated in two steps. In the first one, all the locks in the set L are added to LT(i). Addition of a lock x at one site but not in all does not violate assertion 1 since, x is, by assumption, a member of the set L and therefore is present at every site. The second step is the removal from LT(i) of all the locks in the set R. Removal of a lock from a LOCK table at a given site still makes it present at this site. Since, by assumption, the LOCK table has not yet been updated at all sites, the locks removed from LT(i) are in the LOCK table of at least one site and are present at all sites. Thus, assertion 2 is also valid and the proof is complete.

Now, it remains for us to analyze the transparent failures. Those are all the failures other than the newLC crash already discussed. We can have either a process or processor failure which simply knocks out one of the sites in the component, or the component can

be partitioned into two or more components. In either case, a set of one or more nodes are isolated from the set of nodes which participate in the LCR mechanism. The nodes in this set will not be considered any more for the rest of the LCR algorithm. However, we have to show that no inconsistencies are generated by a node dropping out during the execution of LCR.

For this purpose, we will examine all the possible instants at which a node j may crash.

CASE 1: during the 'nomination phase'

Here we have to show that the sets L and R will not be perturbed by any contributions already made to them by node j. Node j can crash at three possible instants.

CASE 1.1: before the NA message first reaches it.

In this case node j is simply removed from the cycle without contributing to the formation of either L or R.

CASE 1.2: after the NA message reaches it and before it is forwarded to the next node in the sequence.

Here, the node which sent the NA message to node j will timeout, detect its crash and send the NA message to the node which follows node j in the sequence. Again no contributions have been made to the sets L or R.

CASE 1.3: after the NA message has been forwarded

A crash of node j at this point is equivalent to a crash of a node during the 'LOCK table update' phase since node j already played its role in the 'notification phase'. Therefore, this case reduces to the next one to be examined. The reader should notice that the robustness of this recovery mechanism relies heavily on the fact that elements are only added to the sets L or R if the appropriate requests are present at all sites (intersection approach) as opposed to considering requests which are present in at least one site (union approach).

CASE 2: during the 'LOCK table update phase'

A crash of a node during this phase will have no effect upon other nodes, resulting only in the removal of this node from the up list of the logical component which is recovering

Examination of all these cases completes this proof. []

The above result allows us to relax the assumption made in assertion 4 that no additional crashes occur during LCR and state the following assertion.

ASSERTION 5: At the end of the LCR mechanism, all the copies of the LOCK table for a logical component are identical to the value that the LOCK table of the crashed LC would have if all the globally accepted requests were allowed to complete right before the crash of the LC.

Finally we prove that every logical component is internally consistent.

THEOREM 3: Every logical component is internally consistent

Proof: Let C be any logical component. We have to prove that:

i) the set of LOCK tables of C is LT-consistent
ii) there is one and only one LC for C.

Statement i) is clearly true for normal operation of component C since assertions 1 and 2 were demonstrated for this case. Now, by assertion 5 all the copies of the LOCK table are identical at the end of LCR. So, in this case LT-consistency is trivially satisfied.

Statement ii) was proved to be correct in assertion 3 for the case in which no additional crashes occur during LCR. But, by theorem 2, LCR is robust. This allows us to consider the effect of LCR as if no additional crashes occur during its execution, and concludes the proof [].

## 3.3 - Single Node Recovery

So far we have described how the system recovers from a logical component crash. We show now how a node which is down becomes active again, or in other words, how it gets logically connected to a logical component. Let node j be such a node. The first step to become active is to find out who is the LC. This step is carried out by sending the "WHO IS THE LC ?" or WLC message to any up node. Then, node j sends a message called "HI THERE" or HT to the LC telling him that node j is alive again. If the LC is not undergoing any kind of crash recovery it will send its LOCK table and its up list to node j. An "ACCEPT LOCK" or "ACCEPT RELEASE" message is sent to node j by the LC for every lock or release lock request for which not all the LA or RA messages have been received.

## 3.4 - Robustness of SNR

THEOREM 4: The Single Node Recovery (SNR) algorithm is robust.

Proof: Let j be the recovering node and let LCi be the LC to which node j is trying to connect with. The proof is extremely simple since the only two crashes of interest are: a) LLCj crash and b) LCi crash. Case a) is clearly a terminal case. Case b) is also a terminal crash since a crash of LCi, before it is able to send the LOCK table to LLCj, prevents the LOCK table from being received by node j, thereby implying in SNR having to be restarted. This completes the proof. []

## 3.5 - Logical Component Merge

As a result of the Logical Component Recovery algorithm an LC will be elected in each logical component of the network. Transactions which are local to a component will continue to be serviced as if no disconnecting crash had occurred. On the other hand, transactions which span more than one

component will have to wait until the components involved are brought together again. It is the responsibility of each LC to detect when two components are physically connected again and to take the necessary steps to merge them into one logical component. The merge of logical components will always be done on a pairwise basis. The whole Logical Component Merge mechanism is divided into two phases, namely a 'reconnection detection' phase and a 'merge' phase.

In the 'reconnection detection' phase, each LC sends periodically a "WERE YOU ALIVE" or WYA message to every node not in its up list. The purpose of this message is to detect the existence of sites which were not reachable before but which were up. For the purposes of the description that follows, let the two logical components to be merged be called C1 and C2. Let LC1 and LC2 be their respective LCs and U1 and U2 their respective uplists. LC1 will take an active role during the whole recovery phase, while LC2 will take a passive one. As we will see, a crash of LC1 while the recovery mechanism is in progress will result in abort, while a crash of LC2 after the 'reconnection detection' phase is tolerated. Assume now that site #j in C2 received a WYA message from LC1. A component is said to be in NORMAL status if it is not undergoing any kind of crash recovery mechanism. If component C2 is in its NORMAL status, site #j sends a "YES I WAS" or YIW message to LC1. This message carries within it the identification of LC2.

At this point LC1 has to establish a logical connection with LC2. This connection is called a primary-secondary or P-S connection type with LC1 being the primary and LC2 the secondary. Since we require that LCM be done in a pairwise basis, the following conditions must be enforced by the protocol that establishes a P-S connection:

C1: an LC cannot be primary (secondary) for more than one P-S connection.

C2: an LC cannot be primary and secondary simultaneously.

The P-S connection is attempted by having LC1 send a "LET US MERGE" or LUM message to LC2. The status of LC1 is now changed to ATTEMPT. If the status of LC2 is NORMAL, which means that neither Logical Component Merge nor Logical Component Recovery is being attempted, LC2 sends a "MERGE ACCEPTED" or MA message to LC1 and changes its internal state to SECONDARY. Upon receipt of the MA message the connection is considered to be successfully established by LC1. If the status of LC2 is not NORMAL then a "MERGE ATTEMPT REJECTED" or MAR message is sent to LC1 which will either retry later or will try a connection with another LC.

The above interconnection strategy could clearly allow undesirable race conditions to occur, such as having two LCs trying to play the role of primary, leading the system into deadlock situations. To avoid this problem, we assign a site dependent priority to each LC (no two sites have the same priority). LUM messages from lower priority LCs are rejected. LUM messages from higher priority LCs, if received while the connection has not yet been completed, i.e. the MA message has not been received, cause the connection being attempted to be broken. To this end the primary sends a "CLOSE CONNECTION" or CC message to its intended secondary.

That the protocol outlined above satisfies conditions C1 and C2 is proved in section 4.1. Figure 1 shows a state transition diagram describing the interconnection protocol. This

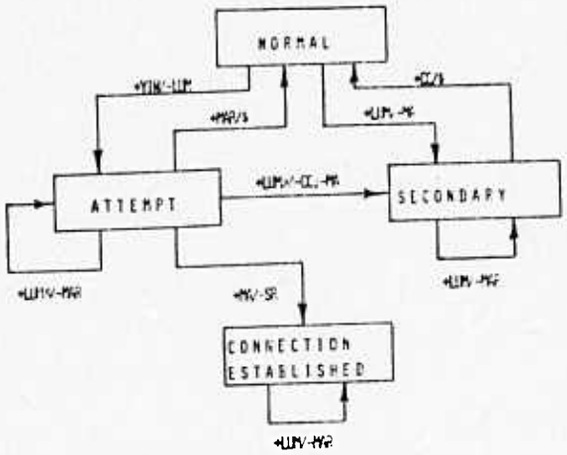


FIGURE 1 - STATE TRANSITION DIAGRAM FOR P-S CONNECTION ESTABLISHMENT. A PLUS (+) SIGN INDICATES RECEPTION OF A MESSAGE AND A MINUS (-) SIGN INDICATES TRANSMISSION OF A MESSAGE. THE SIGN < INDICATES THAT THE MESSAGE IN QUESTION ORIGINATES FROM A LOWER PRIORITY SITE, WHILE > INDICATES A HIGHER PRIORITY SOURCE. THE DOLLAR SIGN ($) INDICATES THAT NO ACTION IS TAKEN DUE TO A STATE TRANSITION.

protocol is the same for every node. Node labels are STATUSes, while arc labels are of the form R/T where R is the message whose arrival triggers the transition and T is a sequence of actions (transmission of messages) which occur as a consequence of the transition.

After a P-S connection has been established between LC1 and LC2, they will not accept any more new lock or lock release requests from nodes in their components and will complete all outstanding ones. An outstanding request is one for which all AL or AR messages have been already sent but not all the corresponding LA or RA messages have been received. After all outstanding requests have been completed by LC2 it sends to LC1 a "READY TO MERGE" or RTM message containing as arguments the uplist U2 and the LOCK table at LC2 which now is the same for all nodes in C2. The receipt of the RTM message by LC1 marks the end of the 'reconnection detection' phase.

The 'merge' phase will construct the union of the LOCK tables at both components. Notice that up to this point no permanent change has been done to any LOCK table, nor up list of any node. LC1 sends a "SUBSTITUTE YOUR TABLE" or SYT message for a cycle through the set of nodes in TEMP_U = U1 U U2. The SYT message is the agent which confirms the merge of the two components by taking within it the new LOCK table for the component. Also, the up lists are updated and LC1 becomes the new LC of

the new logical component.

### 3.6 - Robustness of LCM

THEOREM 5: The Logical Component Merge (LCM) Algorithm is robust.

Proof: The completion point for the LCM algorithm is the point where the SYT message has already been received and accepted by one LLC.

Let LT(i), U(i) and LC(i) be respectively the LOCK table at site i, the up list at site i and the LC identification as known by site i. It is worth observing that changes to the values of LT(i), U(i) and LC(i) at any site i other than the LC-1 site are only done upon receipt of the SYT message.

Let us examine the possible cases of crashes before the completion point:

CASE 1: crashes during the 'reconnection detection' phase

A crash of either LC1 or LC2 in this phase will cause LCM to be aborted and a LCR to be started at the component who had an LC-crash. Since no LOCK table nor up list has been changed so far, this is a terminal crash. Since LC1 and LC2 are the only processes involved in this phase, we conclude that this phase is robust.

CASE 2: crashes during the 'merge' phase

A crash of LC1 during this phase will interrupt LCM and start LCR for component C1. As no permanent changes have been done already, this is a terminal crash. A crash of any other node (including LC2) clearly does not affect any other node nor the mutual consistency of the merged logical component [].

## 4. - Disjointness of the Recovery Algorithms

We show here that there is no interaction between the three recovery algorithms. To that effect one has to show that:

a) LCM is done pairwise
b) LCR, LCM and SNR are mutually exclusive.

To verify condition a) we only need to show that conditions C1 and C2 stated in section 3.5 are satisfied by the P-S connection protocol. This verification is done in section 4.1. Condition b) is shown to hold in section 4.2.

### 4.1 - Disjointness of LCMs

Consider a directed graph G whose vertex-set is the set of LCs and which has two distinct types of arcs, namely e-arcs and a-arcs. There is an e-arc from vertex i to vertex j if there is an established P-S connection between vertices i and j, vertex i being the primary. Equivalently, an e-arc from vertex i to vertex j is said to be created in G whenever vertex i enters the CONNECTION ESTABLISHED state (see figure 1). There is an a-arc from vertex i to vertex j if vertex i is attempting a P-S connection to vertex j. Such an a-arc is created as soon as vertex i enters the ATTEMPT state (see figure 1). The graph G displays the pattern of established and attempted connections. Let e-G be the subgraph obtained from G by considering only e-arcs of G and a-G be the one obtained by taking only the a-arcs.

Conditions C1 and C2 can now be rephrased as follows:

C1.1: 0 <= indegree(v) <= 1 and 0 <= outdegree(v) <= 1 for all v in e-G.

C2.1: indegree(v) * outdegree(v) = 0 for all v in e-G.

Every a-arc will either be deleted from G when the attempted connection is broken or will become an e-arc if the connection is successfully established. So, we want to prove the following:

THEOREM 6: Given a graph G whose e-graph satisfies conditions C1.1 and C2.1, the new e-graph obtained from G as new connections are established also satisfies those conditions.

Proof: It can easily be seen, from the protocol specification, that condition C1.1 is satisfied not only by the initial e-graph but also by the graph G, since:

a) if there is already a connection between vertices i and j or one is being attempted, no new connection is attempted by neither vertex i nor vertex j.
b) if a connection has already been established or is being attempted, the secondary will reject all further attempts.

So, it remains for us to examine all the possible cases in which condition C2.1 could conceivably be violated in G and show that the resulting e-graph obtained when one or more a-arcs become e-arcs still satisfies this condition. There are four possible cases, two of which can never happen due to the protocol specification, while the remaining two have to be examined. Given any three vertices a, b and c, the four possible cases are:

a) (a,b) and (b,c) are e-arcs.
b) (a,b) is an e-arc and (b,c) is an a-arc.
c) (a,b) is an a-arc and (b,c) is an e-arc.
d) (a,b) and (b,c) are a-arcs.

Cases a) and b) are the impossible ones. In case c) the attempted connection between a and b will fail since there is an established connection from b to c (see the self loop at the CONNECTION ESTABLISHED state of the diagram of figure 1). Therefore, arc (a,b) will disappear. In case d) nodes a and b are in the ATTEMPT state. If (a,b) becomes an e-arc we can see that the transition labeled LUM/CC;MA from state ATTEMPT to the state SECONDARY is taken at vertex b, causing the attempted connection (b,c)

to be broken. Therefore arc (a,b) becomes an e-arc while arc (b,c) disappears. On the other hand, if (b,c) becomes an e-arc in the first place we are back to case c) which was already examined. []

We take the opportunity here to prove that the P-S connection protocol is such that all the a-arcs in G will, in a finite time, (of the order of magnitude of the transmission delay time in the network) either disappear or become e-arcs. In other words, the P-S connection protocol is deadlock free.

THEOREM 7: The P-S connection protocol is deadlock free.

Proof: We must prove that there can be no long lasting cycles in G. The interesting case is, of course, that of cycles made out only of a-arcs, since as shown in the previous theorem, any a-arc adjacent to an e-arc will disappear in a finite time.

Consider a cycle in a-G and two adjacent a-arcs (a,b) and (b,c) in the cycle. Vertices a and b are in the ATTEMPT state. There are only two possible cases to consider:

CASE 1: [PRIORITY(a) > PRIORITY(b)]: In this case, if the "MERGE ACCEPTED" message from vertex c is received by b before the "LET US MERGE" message from a then (b,c) becomes an e-arc and (a,b) disappears.

CASE 2: [PRIORITY(a) < PRIORITY(b)]: Here, arc (a,b) will disappear since a has lower priority than b.

In any event, the cycle will be eventually broken. Note also, that vertex c could be the same as a and the above analysis is still valid. []

## 4.2 - Disjointness of LCR, LCM and SNR

We first define a node state transition diagram as a directed graph whose vertices are states of a network node and whose arcs represent transitions between states. The state of a node i is the 3-tuple [STATUS(i), LC(i), U(i)], where LC(i), U(i) are as defined before. STATUS(i) is the status of the component to which site i is attached as viewed by site i. NORMAL status indicates that neither LCR nor LCM is in progress; RECOVERY means that LCR is taking place and QUIESCENT indicates that LC(i) is rejecting further requests. The labels on the arcs specify the conditions upon which a transition between two states occurs. These conditions can either be a crash detection or a message arrival. The diagram, shown in figure 2, shows all possible state transitions for a node, other than LC1, which is in a component C1, with LC equal to LC1 and up list equal to U1. From every state there is a transition to the DOWN state. These transitions are not represented in the diagram for obvious reasons.

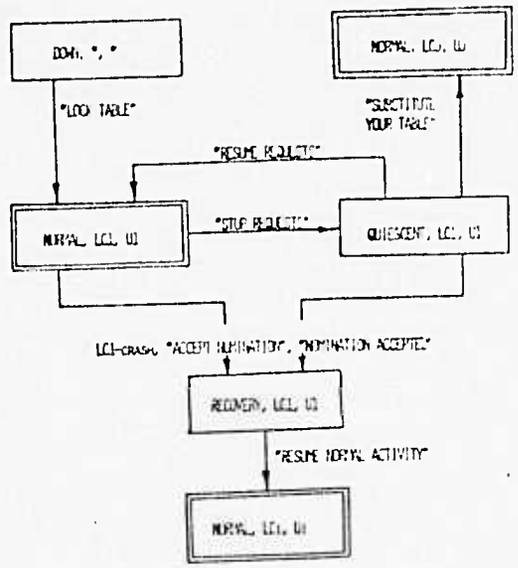


FIGURE 2 - NODE STATE TRANSITION DIAGRAM. THE FOLLOWING RELATIONSHIPS ARE OBSERVED:
- Uj = Ui ∪ U2 WHERE LC2 IS IN Uj.
- Ui IS CONTAINED IN U1.
- LCi IS IN Ui.

The state [NORMAL, LCj, Uj] is state which resulted from a successful merge of component C1 with another component, for instance C2. The state [NORMAL, LCi, Ui] is a state which resulted from a successful Logical Component Recovery.

By inspection of the diagram, we observe that a node can only go from one normal state to a different normal state after one and only one recovery mechanism has been completed. Therefore, there is no interaction among the three recovery mechanisms.

## 5. - Logical Component Mutual Consistency

Let us show here that the CLC protocol (including the recovery mechanisms) is such that the set of Logical Components into which the network is partitioned is mutually consistent.

THEOREM 8: The set of logical components into which the network is partitioned is mutually consistent.

Proof: By theorem 3 each one of the logical components is internally consistent. It remains for us to prove that there can be no lock present at any LOCK table of any component which conflicts with another such lock of any other component. This theorem is trivially true when there is only one logical component. Further net partitioning does not destroy this property since locks are only granted if they are local to a component, which implies that they do not conflict with any other lock granted at any other component. []

## 6. - Database Consistency

We show here that given a deadlock free, consistency preserving locking mechanism for a centralized database (CDB), the CLC protocol can be used to implement an equivalent robust, deadlock free, consistency preserving locking mechanism for a distributed database (DDB). A database is said to be in a consistent state if all the data items satisfy a set of assertions or consistency constraints. A transaction is a sequence of accesses which take the database from a consistent state into another consistent state. Thus, a transaction is the unit of consistency. Let us define an access as the pair (P,a) where P is a logical description of the portion of the database to be accessed and a is an access mode (e.g. read,write,delete,etc.). If all the locks are granted by a process which has complete knowledge of every other active locks (as is the case with the LC) and if every access is checked against the LC copy of the LOCK table (this condition will be relaxed later), to see whether the transaction holds the necessary locks , then the 'lock scheduler' for a CDB described by Eswaran [1] can be implemented in a straightforward manner with the use of the CLC protocol. Such a locking mechanism has the properties of being robust and preserving the consistency of the DB. Notice that deadlock prevention or detection mechanisms can be carried out by the LC since it has complete control over all activities in its component. Recall that if the network is partitioned into more than one component, locks granted in one of them do not conflict with locks active in others. Therefore, distinct LCs manage disjoint sets of "resources", where a resource here means an individually lockable data item in the DB. So, a deadlock prevention or detection policy can be implemented in each LC independently of all the others.

The requirement that every access be checked against the LOCK table at the LC-site can be relaxed in favor of having the access checking done locally. In order for this to be possible a lock must be considered to be active at a given site i for a time interval T2 contained in the time interval T1 during which the lock is active at the LC-site, otherwise some portions of the DB could be locked in conflicting modes for different transactions. Figure 3 shows a double time-axis diagram displaying time at the LC-site and at a given site i where a lock request is originated. T1 starts when the "CONFIRM LOCK" message is sent to every site in the component and ends with the broadcast of the "CONFIRM RELEASE" message. T2 starts with the arrival of a CL message at site i. Although a lock is only removed from a LOCK table when the corresponding "CONFIRM RELEASE" message arrives, it can be flagged as 'waiting for removal' as soon as a "RELEASE LOCK" message is sent from the LC to site i. For access checking purposes, all flagged locks must be considered as non active. The extra precaution that must be taken in this case is to unflag all flagged locks after LCR has taken place.

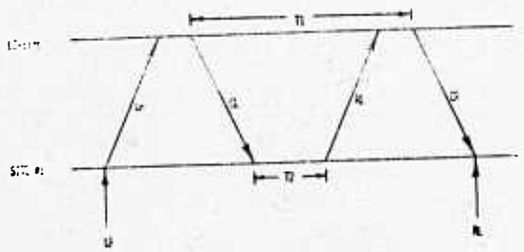

FIGURE 3 - T1 IS THE TIME INTERVAL DURING WHICH A LOCK IS CONSIDERED TO BE ACTIVE AT THE SITE WHERE THE LC IS LOCATED. T2 IS THE TIME DURING WHICH THE SAME LOCK IS CONSIDERED TO BE ACTIVE AT THE REQUESTING SITE #i.

## 7. PERFORMANCE RESULTS

Some of the results of the cost and delay analysis for the CLC protocol [2] are presented here. The update model used in this analysis is such that some of the previously defined messages are grouped into a single physical message. These results indicate that the average update delay, Dupdt, does not depend directly on the size of the network for many network topologies of interest and its expression is given by

$$Dupdt = 2*T + 3*TMAX + W$$

where T is the average message delay introduced by the network between two distinct sites, TMAX is the average maximum delay between a sender and several destinations and W is the average waiting time for a lock request to be granted at the LC.

Lower and upper bounds for the average recovery delay, R, are given by

$$R >= (n+1)*T + 3*TMAX$$

and

$$R < [a*(n-2) + n + 1]*T + 3*TMAX$$

where n is the number of sites in the network and a is the ratio Tout/T where Tout is the time after which a nominator assumes that the nominee is down and sends another "ACCEPT NOMINATION" message to the next site in the nomination order.

The average communications cost, Cupdt, incurred by an update is

$$Cupdt = (3*n - 1)*M$$

where M is the average communications cost per message. Lower and upper bounds for the recovery cost Crec are given by

$$Crec >= (5*n - 2)*M$$

and

$$Crec < (6*n - 4)*M$$

## 8. - Extension

It has been observed in most of the existing distributed systems that a large percentage of the generated transactions is local, in the sense that the resources needed to satisfy a given transaction are either located at the site of origin of the transaction or in neighboring sites. This observation suggests that significant savings in terms of communications cost and delay can be achieved if one optimizes the operation of the

algorithm to adapt to such a highly skewed distribution of activity. To illustrate the point, consider a set of interconnected computer networks. We believe that in such a case, most of the operations will be confined to one computer network while relatively few operations will cross network boundaries.

This section outlines an extension to the CLC protocol that permits the forms of performance optimization needed for the cases discussed above. The extension, which we call an HCLC (for Hierarchical CLC) protocol, consists of a hierarchical organization of resource controllers. A tree of controllers is provided where the root is considered to be at level 0 and all the children of a controller at level i are at level i+1 in the hierarchy.

Each controller (except for the leaves) serves as an LC for its children. Also, each controller (except for the root of the hierarchy) acts as an LLC for its parent. Therefore, each controller has to maintain two distinct LOCK tables, which we call parent-LT and child-LT. The parent-LT for the root controller contains one lock for the whole DB in exclusive mode. The child-LT for a leaf is empty.

An intuitive description of the normal operation of the HCLC protocol can be easily understood in the light of an example. Figure 4

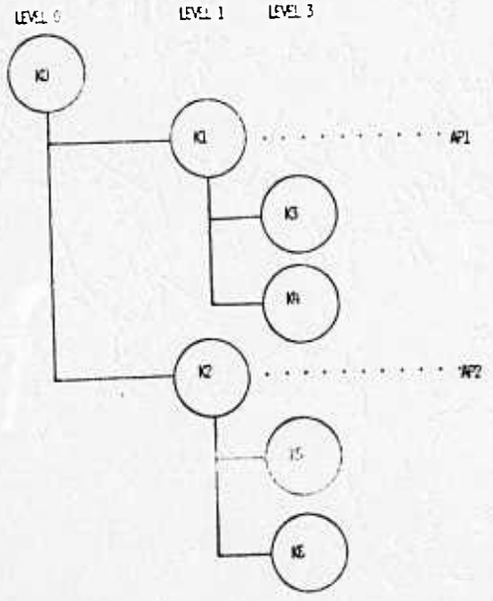


FIGURE 4 - HIERARCHY OF LOCK CONTROLLERS. AP1 AND AP2 ARE APPLICATION PROGRAMS.

shows a three-level hierarchy. Application programs interact with lock controllers K1 and K2 at one level above the leaves (since the leaves are LLCs). This interaction is the same as the AP-LC interaction in the CLC protocol. Actually, application programs are not aware of the fact that the controllers are hierarchically organized. Let a lock request, x, from AP1 be submitted to K1. If x conflicts with any other lock in child-LT(K1) then the lock request is treated in the same way as in the CLC protocol. If there is no conflict, K1's parent-LT is searched for a lock y which covers x. A lock x1 is said to cover a lock x2 if the portion of the DB specified by x2 is contained in the portion of the DB addressed by x1 and if the lock mode specified by x1 is not weaker than the lock mode in x2. The existence of a lock such as y in parent-LT(K1) indicates that K1 currently has control over the resources requested by AP1. If y is found, the lock request x can be granted and to this end K1 interacts with K3 and K4 in the same way as an LC interacts with the LLCs in its component. On the other hand, if y cannot be found, the lock request x is submitted by K1 to K0. K0 will act with respect to K1 and K2 in the same way that K1 did with respect to K3 and K4. The difference in this case is that since K0 is the root there is a lock in parent-LT(K0) for the whole DB in exclusive mode. This lock covers any other lock.

In an HCLC protocol, locks may be released either explicitly or automatically. Locks in child-LT(Ki), for i=1,2, are released explicitly upon request from APs using the same mechanism described in the CLC protocol. Locks in parent-LT(Ki), for i=1,2, can be released automatically as soon as there are no locks in the corresponding child-LTs which depend upon them. To this end, each lock y, in parent-LT(K), for any controller K, has associated with it a list of locks in child-LT(K) covered by y. Also, each lock x in a child-LT(K) points to the lock y in parent-LT(K) which covers x. When a lock x is explicitly released from child-LT(K1) the lock list for its corresponding lock, y, in parent-LT(K1) is appropriately updated. Whenever this list becomes empty, a release request may be automatically generated by K1 and submitted to K0. In general, the automatic release of locks can be propagated up to the root.

This hierarchical protocol can be easily adjusted by policy decisions both to delay such releases, and to establish early locks at higher levels in anticipation of local lock requests. Lock management analogous to LRU-like memory management policies are obvious policy candidates.

For the set of interconnected computer networks, a three-level hierarchy could be constructed as follows. There is one LC per computer network, all of them at level 1. Their children, at level 2, are their corresponding LLCs. Finally, the root is any site acting as a global controller for the entire collection of computer networks.

An interesting property of the proposed extension is that there is always one controller which is able to detect the existence of a cycle in the lock-request graph. This controller is the common ancestor, with the largest level number, to all the controllers where requests in the cycle where originated. In the example of figure 4, the common ancestor to K1 and K2 is K0.

Crash recovery algorithms for the HCLC protocol must include mechanisms to reconstruct the hierarchy, in addition to the recovery mechanisms present in the CLC protocol.

## 9. - Conclusion

This paper outlines what we believe to be a fairly general solution to synchronization issues in distributed systems in the face of asynchronous unplanned failures. The algorithms and protocols for normal operation and recovery are robust with respect to the criteria set up at the beginning of this report. We are unaware of any other synchronization protocols which simultaneously satisfy each of those requirements.

The work is primarily suitable for environments in which the cost, including delay, of sending messages is not high relative to the operations which are to be performed once locking is complete. Locally distributed systems often provide examples of such an environment. Geographically distributed networks also fall into this category if the amount of work to be performed after locking is significant relative to the communications cost.

The protocols are also best suited for usage behavior that cannot be directly characterized in advance. It is assumed that query and update activity will be largely ad hoc in nature - the more general case which has been receiving increasing attention in recent years.

The presentation of any substantial protocol would not be complete without an outline of a proof that the protocol is correct with respect to its desired properties. A significant portion of this document is therefore devoted to that purpose. Further analysis using automated tools is also underway.

In conclusion, these protocols should help demonstrate the practicality of integrated cooperation of activities in distributed systems.

REFERENCES:


1. K.P. Eswaran, J.N. Gray, R.A. Lorie and I.L. Traiger, THE NOTIONS OF CONSISTENCY AND PREDICATE LOCKS IN A DATABASE SYSTEM, Communications of the ACM, Volume 19, Number 11, November 1976.

2. D.A. Menasce, G.J. Popek and R.R. Muntz, A LOCKING PROTOCOL FOR RESOURCE COORDINATION IN DISTRIBUTED SYSTEMS, Computer Science Department, UCLA, Technical Report # UCLA-ENG-7808, SDPS-77-001 (DSS MDA 903-77-C-0211), October 1977.

3. P.A. Alsberg, G. Belford, J.D. Day and E. Grapa, MULTY COPY RESILIENCY TECHNIQUES, Center for Advanced Computation, University of Illinois at Urbana-Champaign, CAC Document Number 202, May 1976.

4. P.A. Bernstein, D.W. Shipman, J.B. Rothnie and N. Goodman, THE CONCURRENCY CONTROL MECHANISM OF SDD-1: A SYSTEM FOR DISTRIBUTED DATABASES (THE GENERAL CASE), Computer Corporation of America, Cambridge, Massachusetts, Technical Report # CCA-77-09, December 1977.

5. S.R. Bunch, AUTOMATED BACKUP, in Preliminary Research Study Report, CAC Doc. 162 (JTSA Doc. 5509), Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1975.

6. C.A. Ellis, CONSISTENCY AND CORRECTNESS OF DUPLICATE DATABASE SYSTEMS, ACM/SIGOPS Operating Systems Review, Volume 11, Number 5, Proceedings of the Sixth Symposium on Operating Systems Principles, November 1977.

7. E. Grapa, CHARACTERIZATION OF A DISTRIBUTED DATABASE SYSTEM, Ph.D. dissertation, Report # UIUCDCS-R-76-831, Department of Computer Science, University of Illinois, Urbana, October 1976.

8. R.H. Thomas, A SOLUTION TO THE UPDATE PROBLEM FOR MULTIPLE COPY DATA BASES WHICH USES DISTRIBUTED CONTROL, Bolt Beranek and Newman Technical Report # 3340, July 1976.

9. W.W. Chu, PERFORMANCE OF FILE DIRECTORY SYSTEMS FOR DATABASES IN STAR AND DISTRIBUTED NETWORKS, Proceedings of the National Computer Conference, 1976, pp 577-587.

10. M. Stonebraker, E. Neuhold, A DISTRIBUTED DATA VERSION OF INGRES, Electronics Research Laboratory, UC, Berkeley, Memo # ERL - M612, September 1976.

11. J.N. Gray, Operating Systems: an Advanced Course, Chapter 3.F: NOTES ON DATA BASE OPERATING SYSTEMS, pp. 394-481, Springer-Verlag Berlin Heilderberg, 1978.

12. B. Lampson, H. Sturgis, CRASH RECOVERY IN A DISTRIBUTED DATA STORAGE SYSTEM, Xerox Palo Alto Research Center Technical Report, 1976. (also accepted for publication in the CACM)

# UCLA SECURE UNIX*

DRAFT

Gerald J. Popek, Charles S. Kline, Evelyn J. Walton

University of California at Los Angeles

## 0. Abstract

UCLA Unix is a wholly new operating system whose architecture and implementation are oriented toward highly reliable security and integrity enforcement while supporting a wide degree of system functionality. The system, now operational, demonstrates that it is possible to provide a convenient, efficient secure operating system on conventional, third generation hardware architectures. This paper reports on the development of UCLA Unix. Much of the discussion is concerned with the software architecture which evolved, since a number of innovations are included with surprisingly little mechanism. The methods employed to build and verify the system are also described, and the impact of the requirement to support fully the standard Unix operating system functionality is discussed.

## 1. Introduction

There has been considerable interest for some time in developing an operating system which could be conclusively shown secure, in the sense that the information stored on behalf of a heterogeneous user population was safely protected from unauthorized access or modification, even in the face of skilled attempts to do so. Ear-

---

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

ly attempts to attain this goal consisted largely of auditing an existing system by attempts at circumventing the controls, and then revising the implementation code to block any successful paths that were found. Unfortunately, this approach failed in producing a secure system, largely because third generation operating systems contain so many errors that "penetration audits" followed by patches inevitably led to a system whose controls were still easily penetrated.

From a viewpoint of principle however, there was an even more fundamental limitation to the early approaches, frequently mentioned; testing proves the presence but not the absense of bugs. Therefore, a more strictly constructive method was required, by which it would be possible conclusively to demonstrate the correctness of the security controls. It was hoped that this goal would result in a much superior system in other respects as well. The experience to be reported here strongly bears out that expectation.

UCLA Unix is a kernel based system architecture developed in a manner by which program verification techniques could be (and have been) applied. The system interface is essentially identical to Unix as released by Bell Laboratories [Ritchie 74], and the software presently runs on DEC PDP-11/45s and PDP-11/70s. The kernel structures and verification procedures, together with the choice of language, provide a powerful means by which the system's security and integrity can be demonstrated and assessed. Support of the Unix interface illustrates the robustness and functionality of the resulting system.

However, the kernel and verification goals imposed significant constraints on the size, complexity and general architecture of the system. The result therefore is quite different from what would have been expected otherwise. Nevertheless, in retrospect, we are unaware of any decision forced by these goals which has not also had the effect of simplifying the system's structure and improving overall reliability and integrity. There has been no significant performance penalty either. The pri-

mary cost in obtaining a secure operating system appears to be found in the care required during design and development.

One important fallout of the system design is considerably enhanced system integrity. Improvement results from the significant reduction in common mechanism operating on behalf of all users, a characteristic that was necessary to make verification and certification of the system practical.

In the next sections we outline the UCLA Unix architecture, together with explanations for the design choices. Verification and the programming language are also discussed, and illustrative examples of the effects of Unix functionality on the system's operation are given.

## 2. Overall Architecture of UCLA Unix

The UCLA Unix architecture contains a number of major modules, whose relation to one another is suggested by figure 1. The kernel should be thought of as an operating system nucleus which provides about a dozen primitive operations callable from user processes. That is, the kernel implements a number of abstract types and the valid operations on each type. It is the only module in the system empowered to execute hardware privileged instructions.

One of the abstract types implemented by the kernel is process. A process contains two address spaces (supervisor and user mode on the large PDP-11s). An operating system interface package resides in one address space. In the other, application code is run. When an application program makes an operating system call, control passes to the o.s. package which interprets the call. If necessary, the package issues kernel calls or uses kernel facilities to send messages to other processes to accomplish the needed action. All such calls or messages are controlled by the kernel. Each process is a separate protection domain. The access rights of the domain

are represented by capabilities: a C-list for each process is maintained by the kernel.

There are several processes that are special, in that they perform system related functions. Overall system security depends on the correct operation of two of them.* One, called the policy manager, is the only process capable of altering protection data, and is thus the site where various security policies may be implemented. Type extensions to kernel objects, including file systems, typically would also be supported here. In the UCLA system, security policy plus suitable primitives for the Unix file system to support protection of individual files are built in the policy manager process. The second, "initiator", process initially owns all terminals (i.e. has capabilities for all of them) and is responsible for user authentication. It tells the policy manager what user is to be associated with a given process.

There is one further process which differs from the typical processes employed for applications programming. However, this one, a scheduler, is not relevant to data security. It contains short term resource management policy for cpu and main memory: process scheduling, page replacement strategies and the like. UCLA Unix is a demand paged system; when a process page faults, the scheduler is informed by the kernel so that an appropriate swap call may be issued at some later time by the scheduler. All of its security relevant actions are accomplished through kernel instructions, however.

Thus in normal operation a user first logs into the initiator. That process then sends a message to the policy manager, who initializes a process for the user and moves the user terminal to the new process by issuing appropriate capabilities. Process initialization as well as normal computation take place within the domain of

---

* One might say they are within the "security perimeter." Their size is not large compared to the kernel described here.

the given process. Additional resource requirements or file activity is accomplished through messages to the policy manager. Process switching occurs whenever a given process invokes the scheduler process, or when an appropriate clock interrupt forces such an invoke. The scheduler can then run whatever process it wishes. Page faults also force an invoke of the scheduler, so that it can initiate appropriate page swapping.

## 3. The UCLA Kernel and Abstract Types

The kernel can alternately be viewed as a basic, stripped down operating system or as an implementor of a number of abstract types, together with the operations on those types. One of its more notable features is the fact that a significant number of facilities, normally found in large systems, are included in it despite its very small size and straightforward structure. The basic kernel consists of approximately 760 lines of Pascal code, not including I/O support. The PDP-11 does not have any channels, so that the functions of channel programs must be written as cpu code. I/O support in the UCLA kernel is composed of two portions: a device independent internal interface of approximately 300 lines, and as many device dependent drivers as are required by devices present on a given machine configuration. These are quite small, and for the UCLA installation, supporting many peripherals, approximately 500 lines of code are required altogether. These numbers are relevant because the entire kernel must be subjected to program verification procedures. Given current verification capabilities, this quantity of code is not unreasonable (assuming a clean structure).

The UCLA kernel implements a fixed number of types, the four listed below. Type extensibility as illustrated by CAL-TSS or Hydra is not provided, although simple extensions are now under way to provide a limited form of this facility. The implemented types, together with the permitted operations, are discussed below.

## 3.1 Processes

The process object is defined to eonsist only of the usual state variables plus one small page. It does not include the process virtual memory. As a result, kernel calls such as Invoke can be quite simple, merely moving data from tables to cpu registers and viee versa. All process relevant kernel calls are controlled by capabilities. It is not possible to issue or reeeive a Notify for example unless in each case a eapability is present in the process' C-list.

The process abstraction has been carefully developed to permit a large number of processes to be alive: 500 on a PDP-11 would not be unreasonable. To do so, it is necessary that very little locked down memory be required per process, despite the fact that there are asynchronous events taking place (such as I/O eompletions and Notifies) which ean occur when all the memory of a process is swapped out. The process must be notified of these events. However, the obvious solution, kernel queues, are undesirable since they increase verifieation diffieulties and lead to overflow problems when queue space is exhausted. The UCLA kernel avoids this problem by a number of methods, including a generalized page faulting strueture and efforts to keep as much per proeess information as possible in swappable pages allocated to the given process. As a result, less than 20 words of main storage must be reserved for an active process.

The operations available for objects of type process are as follows.

a. Invoke

b. Initialize

c. Map-relocation-register

d. Notify

e. Sleep

Invoke moves the state variables of a process into the epu registers, after first saving those of the currently running proeess, mostly into one of that process's

pages. Initialize clears the state variables of a process and creates those few capabilities needed for the process to bootstrapp itself. The Map call is the means by which a process can adjust its own virtual memory. The call sets the mapping between blocks in the process address space and entries in his C-list (which presumably point at pages). Notify is the mechanism by which one process can interrupt a set of other processes, also passing a very small amount of data. Sleep invokes the scheduler.

## 3.2 Pages

Pages are the abstract storage unit supported by the kernel. All pages have a fixed home location on secondary storage, which is not deallocated when the page is swapped into main memory. There are 3 page sizes in the current implementation, with memory frame sizes currently set at sysgen time to minimize kernel complexity. In order to access a page, a process must first obtain a capability for the page. Then the Map call is used to specify where in the process' virtual address space the page specified by the capability is to appear. At that point the process can attempt to refer to the page. If it is in core, the hardware register will be loaded and the reference will succeed. If not, the process will page fault as described in section 3.7. Since each page is a separate object, controlled sharing of individual pages is easily done.

The only operations on pages are:

a. Swap-in

b. Reflect

Swap-in copies the secondary storage version of a page into main memory, changing the name of the object associated with that destination page frame to the new page. The secondary storage copy is preserved. Reflect updates the secondary storage version to match main memory. Neither of these operations gives the caller access to the

contents of the page, so that the operation can be issued by untrusted code.

## 3.3 Devices

I/Os to all devices, including terminals, are controlled by the same capability mechanism as all other operations. However, devices such as terminals are treated as two devices: an input part and an output part. Two capabilities are therefore required to read and write a terminal, but as a result more robust security policies can be supported.

Completion interrupts are handled just like any other process notification. All those processes with capabilities to receive interrupts from the device, and with interrupts enabled, will receive a notification when the device generates it.

The device operations are as follows.

a. Start-i/o

b. Completion-interrupt

Start-i/o initiates all I/Os except swaps. The Completion-interrupt is the hardware generated call which typically signals completion of a previously started I/O. As an entry point into the kernel, it is little different from any other call.

## 3.4 Capabilities

The capability is the basic kernel representation of protection information: which objects a process is entitled to access. Each process has associated with it a C-list containing those capabilities, stored in pages that can be swapped, but which are directly accessible only to the kernel.*

---

* The policy manager is given read access to capability pages so that it need not keep separate track of which capabilities for pages in a file are outstanding. See the discussion of the policy manager for futher information.

Each capability consists of four fields. First is the name of the object to which this capability refers. Second are the access rights provided. Next is a "guess" value which the kernel uses to attempt to quickly find the entry in a kernel table which maps the object indicated by the capability to a physical location. In the case of pages, the guess is the index into the kernel page table to the slot where that page entry last appeared. It in fact may have been moved by subsequent Swaps and Reflects, so if the entry does not match, a search of the table is required. That event is rare however. The last field in the capability is of no relevance to the kernel, but can be set via the Grant call. The Policy Manager uses it to record the file to which the page or device belongs.

The operations on capabilities are quite limited: they can be Granted and revoked. Revocation is accomplished by granting the null capability into the C-list slot that contains the capability to be revoked. Thus there is no means by which processes can directly pass capabilities. While this fact limits what can be done with capabilities, it also greatly simplifies many issues and avoids a number of the criticisms of certain capability systems, especially the danger of not knowing how access to an object has propagated. As a result, the kernel can more accurately be viewed as containing no security policy. All such decisions regarding rights transfer, including initial granting of rights, are made only by the software running in the process which has the ability to issue Grants. The Policy Manager is the only such process in UCLA Unix.

The only operation on capabilities is

a. Grant/revoke

It adds a specified capability to a specified slot in a specified process' C-list. This call is restricted to the policy manager, who implements security policy.

The C-list composes a local name space for the process. This name space has two effects. First, through message exchanges with the policy manager, the user has com-

plete control over which C-list slot contains a given capability, thereby permitting local management over the name space. Fabry [Fabry 74] points out the significant advantages of this facility. Second, kernel names are not visable to user code. Instead, the capability contains that name. Therefore user code, being unaware of the actual object names, cannot use them for a confinement channel.

## 3.5 Types and Operating Systems

Other authors [Schroeder 77] have noted that the usual views of abstract types to be found in programming languages are not quite suitable for operating systems because of finite resources and circular dependencies. In Multics, for example, the process manager depends on the page abstraction, since the manager is contained in pages, while the page manager is a process and hence depends on the process manager. In a revised design for Multics, abstract types are used in a sophisticated, multiple layered manner to solve these problems.[Schroeder 77] However, as noted by Gaines [Gaines 77], the method required need not involve a sophisticated solution at all, and is largely composed of static allocations.

This is the approach embodied in the UCLA kernel. Processes, pages, and devices are neither created nor destroyed. There are as many pages as there is space on secondary storage for them. The number of processes is fixed by the size of the kernel process table. Devices are added at system generation time. This static view is not really a limitation, since the Policy Manager reuses process "bodies" and pages by reinitializing them via kernel calls. Many systems include these size limitations anyway, although perhaps not so explicitly. As a result, the kernel type structure is exceedingly simple, and yet robust enough for fairly general operating system activity, as illustrated in section 6 on Unix Functionality. Further, the entire kernel is small enough to be locked down in main memory, in space removed from page management, blocking circular dependencies.

## 3.6 Kernel Names

The names for kernel supported objects were designed to maintain several important properties with the minimum of mechanism: a) unique names for all objects, b) clear knowledge of object types at all times, and c) avoidance as much as possible of complex name to location mappings, which must be maintained by kernel code if object protection is to be at all meaningful. Since these names are not visable to normal user processes, who see only C-list indexes, considerable design freedom was present. Therefore, names were chosen to represent the home location of the object; a page name consists of the disk device and block number. Hence no disk map need be maintained or interrogated by the kernel.

## 3.7 Paging, Segmentation and Scheduling

UCLA Unix, unlike standard Unix, is a demand paging system. All user disk I/O, including swapping of the process virtual memory space and file activity, occurs via the paging mechanism.*

Page faulting is invisable to all processes except the scheduler, who is notified by the kernel when a fault occurs, so that it can start a swap. There are actually two "faults" involved in accessing pages. The most significant, just described, occurs when a page is not core resident. The other, called a register fault, occurs when the page is resident but the relevant page register is null. This case is handled in a highly efficient way: the user map table is checked by the kernel to see which capability (and therefore which page) is desired. The appropriate value is then placed in the register and user execution continues.

The preceding outline indicates how the UCLA system provides a complete virtual

---

* A logical disk can alternately be treated as a device, and Start-I/Os issued to it. However, a disk treated in this manner cannot also hold pages.

memory and file system with only a simple set of paging primitives in the kernel. This simplicity was achieved by two major decisions. First, the virtual memory facilities were decomposed into that which had to operate correctly in order to maintain the security and integrity of the system (Swap, Reflect, and Completion-interrupt) and the rest of the virtual memory mechanism (page replacement algorithm, interaction with cpu scheduling, etc.). This decision had a significant effect on the system's resulting simplicity. Second, file activity and process memory swapping were combined into one mechanism. In standard Unix, main memory is broken into two areas: one to hold user process images, and the other for I/O buffers. Each area is managed separately. The I/O buffers are replaced in LRU order, while scheduling of process images is handled differently. All disk I/O buffers are the same size, while process images vary. The code used to handle I/O buffers is in large part different from that used to handle the movement of process images, and significant parts of both collections of code are important to the system's security and integrity.

In UCLA Unix, only one mechanism, paging, exists, and much of its support has been moved out into a scheduler which can not affect the integrity of the system. As explained earlier in the section on capabilities, the user domain also carries some of the responsibility for virtual memory management. By placing some of the responsibilities in the domain for which the action is being taken, error propagation is further limited. Application code is of course unaware of that responsibility, since the o.s. interface is performing the task.

## 3.8 Firmware Implementation

The UCLA kernel has been developed to be a candidate for firmware implementation. To be practical, it is helpful if each call behaves as much as possible as a separate instruction, with no need to be interrupted in execution, nor to issue I/O calls for which the results affect the instruction's behavior, since I/O is typically

slow relative to microprogram cycle speeds. These criteria are met by the UCLA kernel. Therefore it differs significantly from architectures such as Multics or related work.[Millen 76][Organick 71] In both of those systems all of the operating system, including inner rings in Multics and kernel software in the case of Mitre, must be considered as part of the user process. Any process can be suspended in the middle of execution in the inner ring or kernel mode, respectively. Neither of those systems lend themselves to firmware considerations, the Mitre work because of the architecture, and Multics because of its size and architecture.

## 3.9 Verification Impacts

Verification of a full scale operating system is a multistep process, and the methods employed at UCLA are outlined by Popek [Popek 78], with more detail available from Kemmerer [Kemmerer 78]. The effect that the verification and certification goals had on the system architecture was exceedingly positive. Often a design choice presented itself, without any clear basis for resolution except maximizing verification case. In retrospect this criterion was quite effective in making decisions and avoiding design pitfalls. Further, when it became clear subsequent to implementation of certain parts of the system that verification would be difficult, those portions were redeveloped. A good example of this case is outlined in section 3.9.2 below.

### 3.9.1 Sequential Code

The current state of verification tools do not permit proof of parallel programs. Since semi-automated aids are in our view essential, this constraint implied a kernel design and implementation in which each call ran from start to completion without interruption, including the interrupt handlers. The UCLA kernel is built in this way, and so most of it can be proven by standard verification methods.

The cost of this design choice results from delayed servicing of interrupts

which arrive while a kernel eall is in progress. To minimize this problem, eaeh call is designed to run very quickly: approximately one millisecond or less. To do so, no kernel call may do 1/O of its own while in the midst of execution, since virtually all devices respond rather slowly relative to this criterion. While millisecond delays in interrupt servicing may not be suitable for heavy real time aetivity, it appears quite acceptable for interactive systems, which is the nature of Unix.

### 3.9.2 I/O Interface

The PDP-11 does not have any significant channels; instead the device registers are wired into physical address locations and "channel" functions are executed by cpu code. Since all deviees address main memory (and secondary storage) in terms of absolute addresses, I/O management is therefore necessarily a kernel responsibility. That is unfortunatc, for several reasons. First, device semantics are quite complex and difficult to interface with the semantics of the programming language in which kernel code is written. Next, deviees are probably the single largest sourcc of changes to the kernel, since as new types of devices are added, additional verified kernel eode is required to manage the device's actions. To minimize the impact of these problems, kernel 1/O eode was redesigned to provide a device independent level of I/O abstraction within the kernel. Code above that level is not concerned with any of the deviee details. Code below it implements device dependent issues, including any device dependent protection controls. The I/O abstraction level appears similar to a channel interface, with well defined opcodes and operands.

This I/O abstraction level is quite important, likely more so than the process abstractions mentioned by other authors, since at least half of the operating system kernel is eoncerned with I/O.[Schroeder 77][Millen 76] As a result of its use, device semantics have been isolated to the low level drivers. See Walker for more information.[Walker 77]

## 4. The Poliey Manager

The Poliey Manager is the major seeurity relevant process in UCLA Unix. It is responsible for implementing a shared file system, for maintaining whatever security policy is to be supported by the system, and for part of the aetion of proeess initialization, which oeeurs every time a Unix fork operation takes place. Each of these issues is discussed below. Long term resouree alloeation can also be implemented in this process, but currently is not.

### 4.1 The File System and Proteetion Poliey

User eode must see a file structure which is identieal to the Unix tree of directories. However, one should not immediately conclude that the entire directory structure and other file support should be implemented in trusted eode. In faet, one ean make the following argument, largely independent of the seeurity poliey to be enforced.

Most eode to be run in the user domain strictly should not be trusted to be eorreet, at least not to the same standards as the verified secure kernel and policy manager. However, all names, including file names, are either issued, interpreted or transmitted through that eode. Therefore it makes little sense to verify the directory naming seheme of a file system when signifieant amounts of unverified eode issue the names or are in the path leading to the file system. The best one ean do, it appears, is to provide the user with a reliable means to specify a process profile which eharaeterizes the categories of files to which the proeess is to be allowed aeeess. Profile speeifieation and alterations, together with the assoeiation of labels with the file on whieh eategories are based, must therefore be done in a guaranteed reliable way if the verified protection and integrity of the entire operating system is to have any meaning. That necessary secure terminal facility is diseussed in section 7 below.

The file protection labels provided in UCLA Unix consist of a very large variety of "colors". Each file can be labelled with some number of them. Each user (principal in Saltzer's terminology [Saltzer 75]) has a fixed color list associated with him. It is understood that a user potentially can access a file only if his color list covers that of the file. The actual profile for a running process can be set to any subset of the user's color list. There is a separate profile for read and write.

Since there are a large number of colors, many of the usual protection policies can be implemented using them. Public files are labelled with the color public and all users have that color in their list. Denning has noted that military security policy is essentially a lattice, and that the relations of sets and subsets provides just the lattice required. Individual file names are had by assigning a given color to a single file. This color system is still evolving as experience is gained with the user protection interface, especially in the area of control over changes to color lists. Additional detail is provided by Urban [Urban 78].

Given the preceding view of file system protection, one can profitably decompose its implementation into two parts, one a common mechanism relevant to security and integrity, the other executable in the domain of the requesting user process. The common mechanism can support a simple, flat file system. Files are the only significant data type, and a color list is one of the attributes of a file. The simple file system mechanism must include complete space management: disk free lists and maps specifying which pages belong to which files, together with software to manage these data structures.

Many of the facilties normally thought of as part of the file system can be provided by software in the individual process domains as part of the o.s. interface: directory structure, maintenance, and searching; end of file indicators and other file status information such as usage locks. Directories are then contained in files, and access to directories is controlled in the same way as access to any other

files. Assuming that the common mechanism in the policy manager is verified correct, users can affect one another only through the use of files to which they share access.

## 4.2 Process Initialization and Forking

The policy manager must also be involved when new processes are created, since a kernel process body must be Initialized and appropriate capabilities need to be granted to the new process. As much as possible however, one wishes process bootstrapping to take place within the domain of the new process. In UCLA Unix, the normal procedure for process forking is as follows. The requesting process sends a message to the Policy Manager requesting the new process as a member of the same user family. The Policy Manager records the user to be associated with the new process and issues a kernel Initialize call, which zeroes a process body, grants two capabilities to that process, and sets the program counter and status to standard values. The capabilities point to a standard boot code page and a scratch data page respectively.* A third capability is granted by the policy manager upon process request to give the process the ability to communicate with its forking parent. From here on, initialization takes place wholly in the domain of the new proces. The process begins by attempting to execute its boot code, which may cause a page fault. These are handled normally. Eventually the boot code will load the o.s. interface and presumably a Unix Shell into its address spaces.

## 4.3 Other Policy Manager Responsibilities

In UCLA Unix, the Policy Manager is also responsible for control over access to the other kernel supported objects besides pages: processes and devices. Devices ap-

---

* The boot code is actually the Kernel Interface SubSystem discussed in section 5.

pear as speeial files and inter-process communieation takes plaee through pages whieh appear as part of a file. Therefore, colors are uniformly employed for access eontrol in these eases too.

An ARPANET eonnection is provided in UCLA Unix; aceess to it must be controlled and support for initial network eonneetion activities is required. Aceess eontrol is done by making each host a special file and using eolors. See seetion 8 below for a discussion of initial eonnection protocols.

## 5. The Kernel Interface SubSystem

Sinee the kernel is an operating system nueleus of minimum size and eomplexity, one ean properly expect that it is not a convenient base to build on. Traditional systems provide a good deal of "extension" for eonvenienee. While at first glanee the o.s.interface has this responsibility, it should be noted that a eonsiderable amount of code is written to run directly on top of the kernel: the o.s. interfaee, the network manager, proeess initialization, and the seheduler, for example. Each of these need basically the same extensions: eapability management, inter-process eommunieation support, virtual memory eode, and some file system interfaees. Therefore we have developed an intermediate interface between the o.s. interfaee and the kernel. The software whieh implements it provides a much more convenient interface to the kernel and is ealled the Kernel Interface SubSystem (KISS). As an extension mechanism, the KISS manages the entire environment of the process. In general, no other code in the process makes kernel ealls, sends messages to the seheduler or policy manager, etc. Thus this software package has primary responsibility for maintaining a convenient "virtual maehine" for the user proeess.

The KISS of eourse runs as part of the user proeess domain, and is architecturally contained in the same address space of the process as the o.s. interface. The KISS can be viewed as an inner ring in the sense of Multies, and if appropriate

hardware were available, that would be an effective means of implementation.

## 6. The Unix Interface

The operating system interface has the responsibility of providing a user program interface which is as much as possible identical to standard Unix.* It handles user system calls either by performing them itself if possible, or making the appropriate kernel calls or service requests to the policy manager to get the desired action accomplished. Much of the Unix o.s. interface is actually lifted from the standard Unix operating system. Most of the changes consist of wholesale deletions of functions, resulting from the fact that many of those functions are redundant given the available kernel facilities and the fact that the o.s. interface is essentially a single user system. All scheduling support could be removed, since scheduling is done in a separate process. A more drastic change concerns I/O buffering. In standard Unix, buffers contain significant structure to aid in multiuser and LRU operation. In UCLA Unix, most of that function disappears since it is done by the paging mechanism supported by the kernel and scheduler. I/O support is replaced in the o.s.interface by code that requests file opens and relevant page capabilities from the Policy Manager, and issues Map calls to add those pages to the interface's virtual memory. Then the interface merely tries to reference data on the page to move it to the user, and the usual page faulting and swapping action takes place.

New code in the interface largely consists of the KISS, changes to the interface/KISS boundary, ipc support, and maintenance of the process hierarchy. This last issue is discussed below.

---

* There are certain actions possible in standard Unix which will be blocked by the security policy of the secure system.

## 6.1 The File System

The Unix interface has a significant portion of the responsibility for making the user view of the file system equivalent to standard Unix. This task consists of all directory support, including searching, working directory control and the like. Once the desired logical file name is found in a directory, a file open request of the policy manager can be made using that name.* Directory searches are done by first opening the containing file, like any other. It is the responsibility of the Unix interface to manage its open files in such a way as to keep the working directory open most of the time to minimize search costs.

## 6.2 Forking and Process Hierarchies

In standard Unix, a given user can have a process family active for him. The family is hierarchical in the sense that parents have certain rights over children. However, intra-family protection is not really effective, since any member of a family can convince any other member to destroy itself, and to take other undesirable actions, via standard Unix functions.

Therefore process hierarchies should not be supported by kernel code, and so in UCLA Unix, members of a process family cooperate among themselves to effect family behavior. Of course, the support for process families is provided in the o.s. interface, so that user software need not be concerned. This design choice simplified the kernal, and in light of the observations made above, had little or no effect on the actual protection functionality provided.

In the implementation, each process of a family has a capability for a shared page, set up by family members. In that page, data structures are maintained by the

---

* The logical file name is essentially an inode number.

o.s. interface so that intra-family relationships are properly supported. In doing so, the kernel notification facility is used to great advantage. Unix typically performs a great deal of "one to n" notification: one process issuing a signal intended for the rest of the family. The kernel Notify call is designed to support this behavior efficiently, as well as to be adaptable for other uses.

## 7. Secure User Interface

In order for any user to have assurance that the protection controls of a system are operating in the manner desired, it is crucial that he be sure of the values to which protection policy data has been set. Further, when login takes place, there is an issue of mutual authentication: the user wishes to be sure that he is interacting with the secure system interface, not some clever user simulation of it which collects passwords. For both of these reasons, UCLA Unix contains a small dialoguer process to which the user terminal can be reliably connected. The user causes his terminal to be switched to the dialoguer by typing a predefined sequence of break characters.* The kernel supports the terminal switch through maintenance of terminal modes. A terminal can be thawed or frozen. Capabilities are granted by the Policy Manager giving access to terminals only when thawed, or only when frozen. When the break sequence is detected, or when a line drop occurs, the line is marked frozen. The Policy Manager grants frozen access only to the dialoguer, thawed access in all other cases. In this way, the user can move his terminal to the dialoguer, accomplish whatever change is desired, such as changing process profiles, and then move the terminal back, all without disturbing the state of computation of the process at all so that it can be continued.

---

* Kernel recognition of the break sequence is not expensive since PDP-11 hardware requires character by character terminal input handling anyway.

## 8. The Scheduler

Whenever it is time for a process invocation decision to be made, the Scheduler is invoked, either directly by a user process (i.e. when it wishes to sleep) or by a clock interrupt. The kernel posts a considerable amount of data to the scheduler process, so that it can make sophisticated resource allocation decisions, about both memory and the cpu. Centralizing both classes of resource control permits effective coordination of allocation decisions and therefore potentially higher performance. A large class of scheduling policies can be implemented in this process. Some of them have confinement implications but provide better performance potential than those which do not. This architecture permits the system operator to make the confinement/performance tradeoff, since there is no kernel effect from scheduling policy changes.

The one potential drawback of a separate scheduler process is that it doubles the actual number of process invocations over what is really needed. This overhead is of little consequence if context switches are relatively cheap, and this will be the case for UCLA Unix.*

## 9. Secure Computer Networks

When security is of concern in a computer network, encryption of the lines is generally a necessity, because those lines are not considered safe from tapping or spoofing. However, the usual approach is to encrypt and decrypt the data external to the central machine and its operating system.

---

* Context switches on the PDP-11 are in general fairly slow. Therefore, the scheduler is actually to be run, still as a separate process, in kernel mode of the hardware. This avoids the necessity of extensive state saving and restoring, but requires the scheduler to be written in a language for which it can be demonstrated that kernel data structures are not touched. The implemented scheduler is written in UCLA Pascal. Moving it into kernel mode was not yet complete when this paper was authored.

It should be recognized that the software resident within the operating system responsible for managing the network is both complex and relevant to security and integrity. In standard Unix with an ARPANET Network Control Program (NCP), the NCP, operating as a common mechanism, is of comparable size and complexity to the whole operating system.* Typically, one wishes to protect each network connection separately from each other connection, but the NCP manages them all, including moving data from user buffers through the NCP and out to the network interface device.

Given the availability of a secure operating system, one can entertain the idea of extending the "ends" of the encryption path deep into the operating system. For example, the user process, as it hands data over to the NCP, could be forced to cause the data to be encrypted, so the network software is treated merely as part of the insecure transmission channel. That data would not be decrypted until the receiving NCP handed it over to the destination user. If each connection were encrypted with a separate key, then NCP errors and misdelivery within the host operating system would not affect security. If suitable error correction is incorporated with the encryption, then integrity problems can also be detected.

The main problem in this approach is the initial connection establishment protocol: how to permit users to supply the NCP with parameters telling which site and what type of connection should be established, without large confinement channels in the system. For a discussion of these and related issues, see Kline [Kline 78]. The method of solution outlined there has been implemented in UCLA Unix. The additional kernel code to support secure network operation was quite small. Further, most of the original NCP was kept unmodified, although its lower level was altered to match the kernel interface.*

---

* The NCP being considered was developed at the Univ. of Illinois.
* The Illinois NCP "kernel" was rewritten.

## 10. Programming Language Issues

The programming language employed in software development is usually recognized to have a significant effect on that effort; however when the goal of development includes verification, the effect is heightened. The specific language issues break down here into two groups: those concerned with systems programming, and those concerned with the scale of the verification steps.

Systems programming issues arise in the same way that they occur in most high level systems programming languages. It is necessary to be able to express details of the hardware in the high level language, such as interrupt vectors, hardware device registers, or special instructions. These facilities must be available in the programming language, but in a way that minimizes the effect on the semantics of the rest of the language.

Virtually all the security and integrity relevant code in UCLA Unix is written in a slightly altered Pascal. Obvious verification problems were removed from the language, such as pointers, variant records, and various sources of aliasing.[Lampson 77] I/O facilities were also deleted, since we were building I/O mechanisms, among other functions. The runtime package needed to support Pascal I/O would have been useless baggage, and since it typically would be written in assembly code there would be little chance of ever verifying properties of its operation.

It was also necessary however to add features to Pascal to permit systems programming, as remarked above. Very few additions were actually necessary, and were limited to the following:

a) the ability to declare a variable to be stored at a fixed physical location (to initialize interrupt vectors, access device control registers, etc.),

b) assembly language procedures (so that special hardware instructions could be expressed as a procedure call),

c) the ability to have procedures which take array parameters whose length is determined at call time (to remedy the most significant limitation of Pascal).

We also developed an extensive library system to support independent compilation of program modules, and yet force type integrity across module boundaries. The compiler and library system force recompilation of modules when needed for compatibility with another module which has been altered. This facility is needed since the verification work depends on type enforcement. The language, compiler, and library system are discussed by Walton.[Walton 76]

There are many issues concerned with the scale of the verification effort. It is believed that over half of the original verification effort could be avoided if the language contained more reasonable controls over aspects of program behavior. One of the more obvious examples concerns the integrity of global variables. An important portion of the assertions to be verified state that most of the kernel variables have not been altered by the routine being considered. (After all, much of the statement of security concerns what is not to happen.) These assertions, in the form of a large invariant, could be simply handled by scope controls in the language, such as the Import/Export lists of Euclid [Lampson 77]. Then compile time enforcement could be employed and the verification task correspondngly simplified. UCLA Pascal has been modified to provide Import Lists.

Another example where the verification task can be eased concerns array bounds checking. In Pascal many subscripts can easily be out of range, and therefore potentially reference data other than the given array, violating type rules. There are four reasonable ways to deal with this problem: Subscript checking could be done by hardware, by runtime software generated by the compiler, by runtime software explicitly inserted by the programmer, or it could be verified in many cases that subscripts do not get out of range. The PDP-11 hardware base does not provide any reasonable way to itself check subscript references.* The UCLA Pascal compiler does not

implement array checking code. Therefore a combination of the remaining choices were taken. The resulting assertions which need to be proven compose a significant fraction of the total verification to be done. Clearly here is a fertile area for language support or enhanced verification tools.

## 11. Architectural Observations

UCLA Unix comprises the first verifiably secure, full functionality operating system with a fine grain of protection. The experience gained in its design and development lead us to several conclusions. Most obvious, secure operating systems are feasible to develop, although the development cost is likely to be considerably greater than if higly reliable security and integrity were not such a serious goal. However, the result is a system which appears to exhibit considerably enhanced reliability and integrity, and because of the strict modularity, is easier to modify. Performance does not appear to be adversely affected by the architectural constraints imposed by the various goals. That is, the net result of the security goal seems to be a better system in general.

It should be noted however that one of the central ideas to the success of the work, kernel structured architectures, requires considerable rethinking of the usual operating system architecture views if it is to be effectively employed. Much of the standard operating system wisdoms must be reexamined, or the result will be a "kernel" that is in fact overly complex and not suitable for a rigorous demonstration of correct security and integrity enforcement.

In conclusion, it appears that the goal of obtaining secure operating systems, at least for centralized, medium scale machines, has been largely reduced to (high

---

---

* The new, upward compatible DEC VAX/780 does.

quality) engineering, with the most significant progress required in program verification.

## 12. Bibliography

Fabry, R., "Capability Based Addressing," Communications of the ACM, Vol. 17, No. 7, July 1974, pp. 403-412.

Gaines, R. S. Private communication, 1977.

Kemmerer, R., "Verification of the UCLA Security Kernel: Abstract Model, Mapping, Theorem Generation and Proof," PhD Thesis, UCLA Computer Science Department, 1978 (forthcoming).

Kline, C. S., "Protection Mechanisms for Operating Systems and Networks," PhD Thesis, UCLA Computer Science Department, 1978 (forthcoming).

Lampson, B. et.al., "Report on the Programming Language Euclid," SIGPLAN Notices Vol. 12, No. 2, February 1977.

Millen, J., "Security Kernel validation in Practice," Communications of the ACM, Vol. 19, No. 5, May 1976 pp. 243-250.

Organick, E., "The Multics System, an Examination of its Structure, MIT Press 1971.

Popek, G., and D. Farber, "A Model for Verification of Data Security in Operating Systems," Communications of the ACM, 1978 (to appear).

Ritchie, D. and K. Thompson, "The Unix Timesharing System," Communications of the ACM, Vol. 17, No. 7, July 1974, pp. 365-375.

Saltzer, J., and M. Schroeder, "The Protection of Information in Computer Systems," Proceedings of the IEEE, 1975.

Schroeder, M., D. Clark, J. Saltzer, "The Multics Kernel Design," Proceedings of the Sixth Symposium on Operating Systems Principles, W. Lafayette, Indiana, Nov. 1977.

Walker, B., "Verification of the UCLA Security Kernel: Data Defined Specifications," Masters Thesis, UCLA Computer Science Dept. November 1977.

Walton, E., "The UCLA Pascal Translation System," UCLA Computer Science Dept. Technical Report, January 1976.

Urban, M., "A Policy Manager for UCLA Secure Unix," Masters Thesis, UCLA Computer Science Dept., 1978 (forthcoming).