(1)

Constraints and System Primitives in Achieving Multilevel Security in Real Time Distributed System Environment

Prepared by:
The Federated Software Group, Inc.
342 West Madison
St. Louis, Missouri 63122

Contract Number:
N00039-93-C-0180
Contract Amount:
$48, 880.00
(Competitively awarded)

Sponsor:
Robert D. Patton
PD51E2
5 CPK
Room 535
Arlington, VA 22245-5200

Constraints and System Primitives in Achieving Multilevel Security in Real Time Distributed System Environment

Prepared by:

The Federated Software Group, Inc.

342 West Madison

St. Louis, Missouri 63122

Contract Number:

N00039-93-C-0180

Contract Amount:

$48, 880.00

(Competitively awarded)

Sponsor:

Robert D. Patton

PD51E2

5 CPK

Room 535

Arlington, VA 22245-5200

1.0 Overview

The lack of Multilevel Secure (MLS) computer systems within the Department of Defense (DOD) is recognized as a significant shortcoming because it limits data exchange, assimilation and interoperability. A number of recent world events have reinforced the operational need for MLS systems that permit the rapid flow of time critical data between classified systems and less secure field-level execution systems.

In addition, within complex military equipment, such as submarines or aircraft, the need exists for real-time collection and analysis of MLS data. Since systems such as these involve multiple subsystems, there is an obvious need for distributed operating system capabilities that can function in such environments. The advent of inexpensive, high-performance workstations and multiprocessor systems has also increased the demand for distributed operating system capabilities. Many development efforts currently underway are implementing client-server architectures to take advantage of the enormous quantities of computing power now available on the user's desktop.

Past efforts to build secure systems have resulted in numerous system high computing systems which are limited to a single level of classified data. One result of this has been a proliferation of command and control (C^2) systems, with one system for each classification level of data. Communication between these systems is typically accomplished via one-way electronic gateways or through the use of magnetic media over an air gap, resulting in significant data redundancy and over classification, time delays, and added expense for redundant equipment and operational costs.

The computing environment has also undergone an evolution that has directly affected the construction of secure systems. The movement from a single large centralized host system to a distributed computing environment consisting of multiple workstations communicating through a network has complicated the requirements for MLS computer systems. Use of MLS in a distributed computing environment is not well understood. Of equal concern is the lack of understanding of developing true MLS applications that perform trusted operations. One of the major focuses of several DOD development efforts is the construction of accredited systems that support MLS activities and provide composite views of MLS data. The individual user needs to view all data available to him in a comprehensive fashion, rather than viewing individual sets of UNCLASSIFIED, SECRET and TOP SECRET data that must be mentally or manually assimilated into a single view. For example, if an object such as a flight mission contains data components that exist at different sensitivity levels, the user needs to view the entire data set that comprises the mission as a single view, rather than viewing three separate fragments of data. The problem of data segregation is particularly acute in systems that perform real-time or pseudo-real-time operations.

The types of information processed by the user affects the severity of the data segregation. If the user is in an environment that requires him to analyze documents over an extended period of time, it is perfectly feasible to examine those data as individual, single level sources of information. However, for users making rapid command and control decisions or for instrumentation aboard complex weapons systems, timely assimilation and processing of data at different sensitivity levels into a composite view of the external world is critical.

From the perspective of the systems integrator and the government agency that must fund development of MLS systems, the options are less than ideal. The current security documentation and trusted product evaluation process do not adequately address the certification and accreditation of complex MLS systems composed of many trusted components. The analysis and operational certification of these systems is often the direct responsibility of the designated official of the organization for which the system is being developed.

The development of these systems, partly because the available operating systems and related products lack critical support for trusted applications development, is complex, time-consuming, and costly compared to similar untrusted systems. Ideally, if adequate support for development of trusted applications was engineered into the MLS operating systems, the gap in development cycles between untrusted and trusted systems performing the same functions would narrow.

A number of efforts have been undertaken to develop workable distributed computing environments, such as OSF/1, Plan 9, Chorus, and Amoeba. However, none of these efforts have made serious attempts to address multilevel security. Most security related efforts have been to address identification and authorization of users and processes across a networked environment using custom protocols, methods like those provided by Kerberos, or relying on MLS local area network (LAN) components. Tremendous strides have been made by the Compartmented Mode Workstation (CMW) effort to provide MLS session capabilities and operation using an MLS LAN protocol (MAXSIX). All CMW efforts to date are essentially UNIX-based operating systems that provide MLS sessions and trusted graphical displays along with appropriate identification, authorization and auditing. These products have not yet evolved into any distributed MLS operating environments using heterogeneous processors, although at least one CMW does have some distributed TCB components.

Part of this slow progress is attributable to the lack of understanding of the impact of security on the capabilities that must be provided in a distributed computing environment. For example, the issues of composibility that occur when incorporating multiple trusted components into a single computer environment are poorly understood and not sufficiently addressed in current security guidelines. Problems that a distributed computing environment must overcome - such as global user and device identification and authorization - become even more complex in an MLS environment where multiple trusted computing bases (TCBs) may be combined.

This report is aimed at illuminating some of these problems and identifying the security issues and constraints that must be answered in a MLS distributed computing environment. Much of the effort was focused on studying a number of the currently available systems to see what capabilities are provided, and which approaches might be utilized in developing a distributed, real-time MLS system.

This report does not address all the issues associated with distributed MLS operating systems, nor does it provide a comprehensive solution to the outstanding issues. This report does identify capabilities critical to a MLS distributed operating system by examining the implementation of several related systems.

Unfortunately, no single system provides all the capabilities needed to build a working MLS distributed operating system. Of the systems studied, the variants based on the Mach microkernel (Mach, Real-Time Mach, and Trusted Mach), the Harris CX/RT and CX/SX UNIX operating systems offer the best currently available combination of capabilities. However, the Mach-based variants are currently separate implementations that have not been combined. The Harris systems provide real-time capabilities on a B1 level system but lack distributed operating system capabilities. Also, significant issues remain regarding support for development of trusted applications in these systems. None of the operating systems studied provides adequate support for simplified MLS applications development and accreditation.

One important aspect of the report is the emphasis on designing a secure and functional distributed computing system drawn from our experience in developing the MLS Global Decision Support System (MLS/GDSS) for the United States Air Force. MLS/GDSS is the Air Mobility Command's multi-level secure Command and Control system. So often, the focus by integrators exclusively on security aspects of the system produces a system that is secure but not deployable because it lacks required operational functionality or is unmanageable. Our analysis has focused both on the security issues and constraints associated with a MLS distributed operating system and the impact of these issues and constraints on operational functionality and the ability to develop trusted MLS applications.

During our effort, we have come to realize that many of the problems that exist with the currently available systems are the direct result of insufficient support for the development of trusted applications provided by these systems. As a result of this preliminary study, we believe that a radical approach to trusted operating systems design and applications development support capabilities will minimize many of these problems and provide a true MLS environment in which to develop distributed real-time applications. Much of the problem with the currently available systems is not that they do not provide distributed real-time capabilities, but rather, that the development of trusted MLS applications is so time-consuming and costly it limits the usefulness of the systems.

1.1 Organization of the Report

This report is organized in the following fashion:

- An overview of distributed operating systems and detailed discussion of four modern distributed operating systems: MACH, Sprite, Amoeba, and Plan 9. These four distributed operating system were chosen because they are well documented and represent current approaches to solving problems associated with distributed operating systems. None of these systems provide MLS or real-time capabilities. At least two of these are available in commercial form.

- An overview of the principles associated with real-time operating systems and detailed discussion of four modern real-time operating systems: Maruti, ARTS, RTMACH, and CX/RT. These four real-time operating systems were chosen because they represent current research and COTS approaches to real-time operating system problems.

- An overview of MLS operating systems and detailed discussion of two MLS operating systems: TMACH and SecureWare. These two systems were chosen because they represent a microkernel and monolithic kernel approach to MLS operating system issues. TMACH is based on MACH, a distributed operating system that also has a real-time version under development. Secureware is a popular secure system in use on many CMWs and MLS operating system undergoing formal evaluation. Secureware also encompasses the MAXSIX MLS networking protocol.

- A discussion of several issues related to supporting MLS application development by MLS operating systems and security threats associated with distributed operating systems.

- An analysis of the shortcomings of MLS operating systems with respect to MLS application development. This section draws upon experiences gained in the MLS/GDSS project at Scott AFB over the past four years.

- Finally, a rudimentary listing of requirements for a conceptualized distributed MLS operating system with real-time capabilities. Emphasis in this section is placed on the features that would enhance the use and development of MLS applications on a MLS operating system.

2.0 Distributed Operating Systems

2.1 General Issues

The 1970s were dominated by medium to large sized timesharing systems, typically supporting dozens of on-line terminals. In the 1980s, personal computing became popular, with many organizations installing large numbers of PCs and engineering workstations usually connected by a relatively fast local area network. The basic problem with current networks of PCs and workstations is that they are not transparent to the user. Each user is acutely aware of the multiple machines on the network and typically works only on a single machine. In these environments, few if any, programs take advantage of the multiple CPUs on the network, even though many may be idle. Furthermore, these networks of machines are difficult to manage and present a security nightmare.

In the late 1990s, it is likely that inexpensive, very high performance systems with massive amounts of physical memory will be available to replace current PCs and workstations. In order to better utilize these vast resources, distributed operating systems that are capable of taking advantage of the networked CPUs are needed. In many respects what is needed is to return to the environment of the 1970s where the system resources were transparent to the user except for the terminal screens in front of him.

A number of projects and efforts have developed distributed operating systems for general purpose use. Unfortunately, none have become popular, except in a few confined environments. Certain components of these systems are in general use, such as NFS, but for the most part we still operate in a single networked workstation mode. In the following sections, a number of distributed operating systems are discussed. This discussion is meant to convey some understanding of the issues related to distributed operating system sand to show the various means in which the general problem has been approached. If wide-spread use of any of these systems is a measure of success, then all are probably failures. The one exception to this is Mach, which has made its way into several commercial UNIX variants. Unfortunately, even in these variants, the power of distributed computing is not being exploited to its fullest.

2.2 Analysis of Existing Distributed Operating Systems

2.2.1 MACH [References: 1, 2, 3, 4, 5, 6]

The Mach system is a multiprocessor operating system kernel developed at Carnegie-Mellon University first for distributed systems, then for tightly-coupled multiprocessors with uniform memory access. Mach runs on a wide variety of uniprocessor and multiprocessor architectures, including the DEC VAX system, Sun 3 workstations, IBM PCs, the IBM RP3 multiprocessor, the Encore Multimax, and recently, the IntelParagon. Mach is also supported as a product by a number of hardware and operating system vendors. Mach is the base technology for the OSF/1 operating system from the Open Software Foundation (OSF). Mach separates the Unix process abstraction into tasks and threads. In addition, Mach provides the following:

- Machine independent virtual memory management.
- A capability based interprocess communication facility.
- RPC support.
- Support for remote file accesses between autonomous systems.
- Lightweight user threads known as Mach Cthreads.
- Miscellaneous other support like debuggers for multithreaded applications and exception han[illegible].

Structurally, Mach is organized in a horizontal using micro-kernel technology. The Mach kernel is a minimal, extensible kernel which provides a small set of primitive functions. It provides a base upon which complete system environments may be built. The actual system running on any particular machine is implemented by servers executing on the top of the kernel. The Mach kernel supports the following basic abstractions:

- Task: A task, an execution environment for threads, is a collection of system resources including an address space.
- Thread: A thread is defined as the basic unit of execution. Mach threads belong to the middleweight process class.
- Port: A port is a communication channel and is similar to an object reference in an object-oriented system. Any object other than messages may be represented by a port. An operation on an object is performed by sending a message to the corresponding port.
- Message: A message is a typed collection of data objects used for communication between active threads.
- Memory Object: A memory object is a repository of data which can be mapped into the address space of a task.

2.2.1.1 Memory Management in Mach

The Mach virtual memory management system is designed to be architecture and operating system independent. Architecture independence is achieved by dividing the virtual memory implementation into machine independent and machine dependent portions. The machine independent portion has full knowledge of all the virtual memory related information, whereas the machine dependent portion manages the "pmaps", which are hardware defined physical address maps (e.g., virtual memory page tables). The machine dependent portion only contains the mappings essential to run the current programs. All mapping information for any page in the system can be reconstructed from the information in the machine independent portion at fault time. Due to this separation between machine dependent and independent portions, the page sizes in both portions may not be the same.

The main data structures in the Mach virtual memory system are: a resident page table, which keeps track of information about machine independent pages, an address map which is a per-task doubly linked list of map entries, each of which maps a range of addresses to a region of a memory object, a memory object which is the unit of backing storage, and the Pmaps which maintain the machine dependent memory mapping information. Using these data structures, Mach supports large, sparse virtual address spaces and memory mapped files. Mach implements a single level store by treating all primary memory as a cache for virtual memory objects. Mach allows tasks to allocate and deallocate regions of virtual memory, and to set the protection and inheritance of virtual memory regions. Inheritance may be specified as shared, copy or none on a per page basis. A page specified as shared is shared for reads and writes by both parent and child tasks, whereas a page specified as copy is effectively copied into the child's address map.

However, for efficiency, a copy-on-write technique is used. A page specified as none is not passed to the child. The copy-on-write sharing between unrelated tasks is also employed for passing large messages between them. The virtual memory system exploits lazy evaluation, such as Copy-on-write and map-on-reference whenever possible.

Another important feature of the Mach virtual memory system is its ability to handle page faults and to page out data requests at the user level. A few basic paging services are provided inside the kernel. However, a pager may be specified and implemented outside the kernel at user level. This has become particularly important for real-time implementations of Mach and for implementations addressing the needs of specific parallel architectures. The Mach kernel does not have specific support for distributed shared memory. However, distributed shared memory can be implemented using a server on top of the kernel.

2.2.1.2 Mach Interprocess Communication

A port is a kernel-protected entity which is the basic transport abstraction in Mach. Messages are sent to and received from a port. A task gets access to a port by receiving a port capability with send or receive rights. Mach 3.0 also supports send-once rights for ports: these are useful for implementing RPC. It also provides dead names and dead name notifications, which allow servers and clients to detect each others' terminations. Messages are variable size collections of typed data. Mach supports both synchronous and asynchronous message transfers. The copy-on-write technique is employed for large message transfers.

The ports and the messages together provide location independence, security, and data typing. Mach 3.0 supports port sets to let a few threads serve requests for multiple objects. A receive operation on a port set returns the next message sent to any of the member ports. A no-sender detection mechanism allows object servers to garbage collect the receive right and the represented object. The Mach kernel implements messages only within a single machine. However, the transparency of Mach interprocess communication (IPC) allows a user-level server (network message servers) to extend the IPC across a network. Mach also implements various flavors of communication including server-client remote procedure calls, distributed object-oriented programming, and streams.

2.2.1.3 Mach Process Scheduling

The Mach scheduler consists of two parts: one responsible for processor allocation, and the other responsible for scheduling threads on individual processors.

For processor allocation, a user-level server performs processor allocation, using the mechanisms provided by the underlying Mach kernel. The processor allocation facility adds two new objects to the Mach kernel - the processor and the processor set. An application creates a processor set, and uses it as the basis of communication with the server. A server performs processor allocation by assigning processors to the processor sets provided by the clients. Thus, clients have the power to use and manage processors without having direct control over them. A server satisfies a client's requests in strict order in a greedy fashion.

For thread scheduling, Mach uses a priority based time-sharing scheduling technique within each processor set. Mach schedules individual threads without using any knowledge about the relationships among threads. The scheduler maintains a global run queue shared by the processors and a local run queue for each processor. Each run queue is a priority queue of runnable threads. Mach is "self scheduling" - a processor consults the run queue when it needs a thread to run.

2.2.1.4 The Mach 3.0 Micro-Kernel

The Mach 3.0 micro-kernel has evolved from Mach 2.5 by eliminating all the compatibility code for BSD Unix from the kernel. Major changes include the optimization of its IPC implementation (by optimizing ports and port rights) as well as the use of new algorithms for scheduling, IPC, exception, and page fault handling facilities.

The basic facilities provided by the Mach kernel support the implementation of operating systems as Mach applications. The Unix server, for example, is implemented as a Mach task with multiple threads of control managed by the Mach Cthreads package. The emulation library functions as a translator for system service requests and as a cache for their results.

2.2.2 SPRITE [References: 7, 8]

2.2.2.1 Overview

Sprite is an experimental network operating system developed through an ongoing project at the University of California at Berkeley under the direction of Ousterhout. It is part of a larger research project called SPUR ("Symbolic Processing Using RISCs"), whose goal is to design and construct a high-performance multiprocessor workstation with special hardware support for Lisp applications. Although one of Sprite's primary goals is to support applications running on SPUR workstations, the system is also designed to work well for a variety of high-performance engineering workstations. Sprite is currently available on Sun-2 and Sun-3 workstations.

According to the authors, the motivation for building the Sprite operating system came from three general computer technology trends:

- networks,
- large memories, and
- multiprocessors.

In increasingly more research and engineering organizations, computing occurs on personal workstations interconnected by local-area networks, with larger time-shared machines used only for those applications that cannot achieve acceptable performance on smaller workstations. Unfortunately, workstation environments tend to suffer from poor performance and difficulties of sharing and administration, due to their distributed nature. Sprite is designed to make the distributed environment as transparent as possible and make available the same ease of sharing and communication that is possible on large host-based systems.

The second technology trend that drove the Sprite design was the growing availability of large physical memories. Current engineering workstations typically contain 16 to 32 Mbytes of physical memory. It is expected that memories of 100-500 Mbytes will be commonplace within the next few years. Part of the philosophy behind the Sprite operating system is the belief that such large memories will change the balance between computation and I/O by permitting all commonly-accessed files to reside in main memory.

The third driving force behind Sprite was the imminent arrival of multiprocessor workstations. Workstations with more than one processor are currently under development or commercially available. The overall goal of the Sprite operating system is to provide simple and efficient mechanisms that capitalize on the three technology trends.

The technology trends had only a minor impact on the facilities that Sprite provides to application programs. For the most part, Sprite's kernel calls are similar to those provided by the 4.3 BSD version of the UNIX operating system. However, the authors added three additional facilities to Sprite in order to encourage resource sharing:

- a transparent network file system,
- a simple mechanism for sharing writable memory between processes on a single workstation, and
- a mechanism for migrating processes between workstations in order to take advantage of idle machines.

Although the technology trends did not have a large effect on Sprite's kernel interface, they suggested dramatic changes in the kernel implementation, relative to UNIX. This was not surprising, since networks, large memories, and multiprocessors were not important issues in the early 1970's when the UNIX kernel was designed. The Sprite kernel was engineered from scratch, rather than by modifying an existing UNIX kernel as is the case with so many UNIX-derived operating systems.

Some of the interesting features of the kernel implementation are:

- The kernel contains a remote procedure call (RPC) facility that allows the kernel of each workstation to invoke operations on other workstations. The RPC mechanism is used extensively in Sprite to implement other features, such as the network file system and process migration.
- Although the Sprite file system is implemented as a collection of domains on different server machines, it appears to users as a single hierarchy that is shared by all workstations. Sprite uses a simple mechanism called prefix tables to manage the name space; these dynamic structures facilitate system administration and reconfiguration.

- To achieve high performanc: in the file system, and also to capitalize on large physical memories, Sprite caches file data both on server machines and client machines. A simple cache consistency mechanism guarantees that applications running on different workstations always use the most up-to-date versions of files, in exactly the same fashion as if the applications were executing on a single machine.
- The virtual memory system uses ordinary files for backing storage; this simplifies the implementation, facilitates process migration, and may even improve performance relative to schemes based on a special-purpose swap area. Sprite is capable of retaining the code segments for programs in main memory even after the programs complete, in order to allow quick start-up when programs are reused. The virtual memory system negotiates with the file system over physical memory usage, in order to permit the file cache to be as large as possible without degrading virtual memory performance.

Sprite guarantees that processes behave the same whether they are migrated or not. This is achieved by designating a home machine for each process and forwarding location-dependent kernel calls to the process's home machine.

2.2.2.2 The Sprite Application Interface

Sprite's application interface contains kernel calls that are very similar to those provided by the Berkeley versions of UNIX. In fact, large number of traditional UNIX applications have been ported to Sprite with relatively little effort. The Sprite file system allows all of the disk storage and all of the I/O devices in the network to be shared by all processes so that processes need not worr; about machine boundaries. Second, the virtual memory mechanism allows physical memory to be shared between processes on the same workstation so that they can extract the highest possible performance from multiprocessors. Third, Sprite implements process migration, which allows jobs to be off-loaded to idle workstations and thereby allows processing power to be shared.

2.2.2.3 The Sprite File System

Almost all of the modern network file systems, including Sprite's, have the same ultimate goal: network transparency. Network transparency means that users should be able to manipulate files in the same ways they did under time-sharing on a single machine, The distributed nature of the file system and the techniques used to access remote files are made invisible to users under normal conditions. The LOCUS system was one of the first to make transparency an explicit goal and other file systems with varying degrees of transparency include CMU-ITC Andrew and Sun's NFS.

Most network file systems fail to meet the transparency goal in one or more ways. The earliest systems (and even some later systems, such as 4.2 BSD) allowed remote file access only with a few special programs such as rcp. Most application programs could only access files stored on local disks. Second-generation systems, such as Apollo's Aegis, allow any application to access files on any machine in the network, but special names must be used for remote files as opposed to local file names.

Third-generation network file systems, such as LOCUS, Andrew, NFS, and Sprite, provide name transparency - the location of a file is not indicated directly by its name. It is possible to move groups of files from one machine to another without changing their names.

Unfortunately, most third-generation systems still retain some non-transparent aspects. For example, in Andrew and NFS only a portion of the file system hierarchy is shared; each machine must also have a private partition that is accessible only to that machine. In addition, Andrew and NFS do not permit applications running on one machine to access I/O devices on other machines. LOCUS is unique among current systems because it provides complete file transparency.

The Sprite operating system is similar to LOCUS in that it provides complete file system transparency, so that the behavior seen by applications running on different workstations is exactly the same as it would be if all the applications were executing on a single time-shared machine. A single file hierarchy is uniformly accessible from all workstations. Although it is possible to determine where a file is stored, that information is not needed for routine operation. No special programs are needed to operate on remote files as opposed to local ones; no operations are restricted to local files. Sprite also provides transparent access to remote I/O devices. As in UNIX, Sprite represents devices as special files. However, unlike most versions of UNIX, Sprite allows any process to access any device, regardless of the device's physical location.

2.2.2.4 Sprite Shared Memory Address Spaces

Early versions of UNIX did not permit memory to be shared between user processes, except in read-only mode. Each process had private data and stack segments. Since then, extensions to allow read-write memory sharing have been implemented in several versions of UNIX, including System V, SunOS, Berkeley UNIX, and Mach. There are two reasons for providing shared memory. First, the most efficient means to design many applications is to use a collection of processes in a shared address space. Multiple processes are particularly convenient when an application consists of mostly-independent sub-actions while the shared address space allows them to cooperate to achieve a common goal. The second motivation for providing shared memory is the availability of multiprocessors. If an application is to be decomposed into pieces that can be executed concurrently, there must be fast communication between the components. The faster the communication, the greater the degree of concurrency that can be achieved between those components executing on different processors.

Shared memory provides the fastest possible communication and the best opportunity for concurrent execution. Sprite provides a particularly simple form of memory sharing: when a process forks to create a new process, it may request that the new process share the parent's data segment. The stack segment is still private to each process. It contains procedure invocation records and private data for the process. For simplicity, Sprite's mechanism provides all-or-nothing sharing, i.e., it is impossible for a process to share part of its data segment with one process and part with another. This approach allows Sprite to support efficient synchronization primitives.

2.2.2.5 Process Migration in Sprite

In a workstation environment, most of the machines are idle at any given time. In order to allow users to harness this idle computing power, Sprite provides new kernel routines to move a process or group of processes to an idle machine. Processes sharing a given memory heap segment must migrate together. Sprite keeps track of which machines are idle and selects one as the target for the migration. The process has migration is transparent both to the migrated process and to the user. The only noticeable difference after process migration is a reduction in the load of the home machine.

There are several obvious instances where process migration can be used. For example, there are shell commands for manual migration, which allow users to migrate processes in much the same way that the UNIX shell allows users to place processes in background. Second, Sprite has a new version of the UNIX make utility, called pmake. Pmake, like make, carries out the recompilation of programs when their source files change. Whereas make invokes the recompilations sequentially, pmake is organized to invoke multiple recompilations concurrently, using process migration to off-load the compilations to idle machines.

Process migration, which allows processes to be moved at any time, has been implemented in several systems, but is still not widely available. Process migration is a particularly important capability in a workstation environment. If remote invocation is used to off-load work onto an idle machine, and then the machine's user returns, either the foreign processes will have to be killed or the machine's user will receive degraded response until the foreign processes complete. In the Sprite operating system, the foreign processes can be migrated off the local machine.

One of the most important attributes of Sprite's migration mechanism is its transparency, both to the process and to the user. A process will execute and produce exactly the same results when migrated as it would if it were not migrated. Sprite preserves the environment of the process as it migrates, including files, working directory, device access, environment variables, and anything else that could affect the execution of the process. In addition, a migrated process appears to its user still to be running on the user's home machine. It will appear in listings of processes on that machine and can be stopped or killed or debugged just like the user's other processes. In contrast, remote execute utilities such as rsh do not preserve the working directory or other aspects of the environment, and neither rsh nor rex allows a remotely-executing process to be examined or manipulated in the same fashion as a local process. Other implementations of process migration tend not to provide complete transparency to users, although they do provide complete transparency to the migrated processes.

Application programs invoke kernel functions via a collection of kernel calls. The basic flow of control in a kernel call is similar in Sprite to that of UNIX. User processes execute "trap" instructions to switch to supervisor state, and the kernel executes as a privileged extension of the user process, using a small per-process kernel stack for procedure invocation within the kernel. Sprite's basic kernel structure has been modified to provide support for multiprocessor and network operations. The multi-threaded synchronization structure of Sprite allows the kernel to run efficiently on multiprocessors. Remote procedure call capabilities allow the various invocations of the kernel to invoke operations remotely over the network.

Most operating system kernels, including UNIX, are single-threaded. When a process calls the kernel, a single lock is acquired and then released when the process puts itself to sleep or returns to user mode. In these types of systems, processes are never pre-empted while executing kernel code, except by interrupt routines. This single-threaded approach simplifies the implementation of the kernel by eliminating many potential and difficult synchronization problems between processes.

Unfortunately, single threaded kernels are not easily adapted to a multiprocessor environment. With more than a few processors, contention for the single kernel lock will limit the overall performance of the system. In contrast, the Sprite kernel is multi-threaded. Several processes may execute in the kernel at the same time. The kernel is organized in a process monitor-like fashion, with many small locks protecting individual modules or data structures rather than a single overall lock. The multi-threaded approach allows Sprite to run more efficiently on multiprocessors, but the multiplicity of locks makes the kernel more complex and slightly less efficient, since several locks may have to be acquired and released during a system call.

One of the more important design goals of Sprite was to provide a simple and efficient way for the kernels of different workstations to invoke each others' service routines. In Sprite, this is accomplished using a kernel-to-kernel remote procedure call (RPC) facility. The designers of Sprite chose an RPC-based approach over a message-based approach, because RPC provides a simple programming model, i.e., remote operations appear identical to local procedure calls. Also the RPC approach is particularly efficient for request-response transactions, which is the most common form of interaction between kernels.

The implementation of RPC consists of stubs and RPC transport. The two components of RPC, hide the fact that the calling procedure and the called procedure are on different machines. For each remote call, two stubs exist, one on the client workstation, and one on the server. On the client, the stub copies its arguments into a request message and returns values from a result message, so that the calling procedure isn't aware of the underlying message communication. The server stub passes arguments from the incoming message to the desired procedure, then packages the results from the procedure. The called procedure isn't aware that its real caller is on a different machine.

The second part of the RPC implementation is RPC transport, which delivers messages across the network, and assigns incoming requests to kernel processes which execute the server stubs and called procedures. The goal of RPC transport is to provide the most efficient communication possible between the stubs, while ensuring that messages are delivered reliably.

Sprite's RPC transport uses two techniques to gain efficiency: implicit acknowledgments and fragmentation. Since network transmission is not perfectly reliable, each request and response message must be acknowledged. If no acknowledgment is received within a predetermined time, the sender retransmits. To reduce the overhead associated with processing acknowledgment packets, Sprite uses the scheme where each request or response message serves as an implicit acknowledgment for the previous response or request message from that client, respectively. In the common case of short closely-spaced operations, only two packets are transmitted for each remote call: one for the request and one for the response.

The simplest way to implement RPC is to limit the total size of the arguments or results for any given RPC, so that each request and response message can fit into a single network packet. Unfortunately, the maximum allowable size for a network packet is relatively small (about 1500 bytes for Ethernet), so this approach would result in high overhead for bulk transfers. Delays associated with sending a request, requesting a server process, and returning a response would be incurred for each 1500 bytes. One of the most common uses of RPC is for remote file access, which could have presented a performance limitation. Sprite's RPC mechanism differs from other implementations in that it uses fragmentation to ship large blocks of data (up to 16 Kbytes) in a single remote operation. If a request or reply message is too long to fit in a single packet, RPC transport breaks the message into multiple packets which it then transmits in order, without waiting for acknowledgment. The receiving RPC transport reassembles the fragments into a single large message. A single acknowledgment is used for all the fragments, using the same implicit acknowledgment scheme described above. When packets are lost in transmission, the acknowledgment indicates which fragments have been received so that only the lost fragments are retransmitted.

An interesting security aspect of Sprite is that the various kernels on the network must trust each other, and it is assumed that the network wire is physically secure. Given these assumptions, the RPC mechanism does not use encryption, nor do the kernels validate RPC operations except to prevent user errors and detect system bugs. The RPC mechanism is used only by the kernels, and is not directly visible to user applications.

2.2.2.6 The Sprite File System

In designing the Sprite file system for a network environment, the authors were particularly concerned about two implementation issues: how to manage the file name space in a way that simplifies system administration, and how to manage the file data in a way that provides high performance. An important design consideration was to provide easy administration and high performance without compromising users' ability to share files. To users, the Sprite file system is a single hierarchy, just as in standard time-shared UNIX. To system administrators, the file system is a collection of domains, which are similar to "file systems" standard UNIX. Each domain contains a tree-structured portion of the overall hierarchy. The domains are joined into a single hierarchy by overlaying the leaves of some domains with the roots of other domains. As the operating system traverses the components of a file name during name lookup, it must move automatically from domain to domain in order to keep the domain boundaries from being visible to users.

The interesting naming issues include: tracking the domain structure, and handling file names that cross domain boundaries. These issues are particularly interesting in a network environment where the domains may be stored on different servers, and where the server configuration may change frequently. UNIX and most of its derivatives (such as NFS) use static mount tables to keep track of domains; the mount tables are established by reading a local configuration file at boot-time. These systems experience difficulties responding to configuration changes. In the Sprite operating system, prefix tables provide a more dynamic approach to managing the domain structure. The kernel of each client machine maintains a private prefix table. Each entry in a prefix table corresponds to a domain and it provides the full name of the top-level directory in the domain, the name of the server on which that domain is stored, and an additional token to pass to the server to identify the domain. Prefix tables are not normally visible to user processes. In Sprite, as in UNIX, application programs refer to files by specifying either an absolute path name for the file, or a relative path name, which is interpreted as starting from a current working directory.

Sprite's implementation of virtual memory is very traditional. For example, it uses a variation on the "clock" algorithm for its page replacement mechanism, and provides shared read- write data segments using a straightforward extension of the mechanism that provides shared read-only code in standard time-shared UNIX. In order to take advantage of the networked environment and large physical workstation memories, Sprite uses ordinary files for backing storage in order to simplify process migration, to share backing storage between workstations, and to capitalize on server caches. In addition, Sprite provides so-called "sticky segments" and a dynamic trade-off of physical memory between the virtual memory system and the file system cache. These mechanisms have been implemented to make the best possible use of physical memory as a cache for programs and files.

Backing storage is the portion of disk used to hold pages that have been swapped out of physical memory. Most versions of UNIX use a special disk partition for backing storage and manage that partition with special algorithms. In networked UNIX systems, each machine has its own private disk partition for backing storage.

In contrast, Sprite uses ordinary files, stored in the network file system for backing store. This allows a process to share a pre-existing copy of the code on a new machine during process migration. In Sprite, backing files simplify the transfer of the virtual memory image. During migration, the old machine simply pages out the process's dirty pages and transfers information about the backing files to the target machine. If the code segment already exists on the new machine, the migrating process shares it. Pages are reloaded in the new machine on demand, using the standard virtual memory mechanisms. Thus, the process need only be frozen long enough to write out its dirty pages.

The second, and more important, issue in process migration is how to achieve transparent remote execution. A process must produce the same results when migrated as it would have produced if it had not been migrated. It must not be necessary for a process to be coded in a special way for it to be migratable. For message-based systems, transparency is achieved by redirecting the process's message traffic to its new home. Since processes communicate only by sending and receiving messages, this is sufficient to guarantee transparency. In contrast, Sprite processes communicate by invoking kernel calls. Kernel calls are normally executed on the machine where invoked and may produce different results on different machines. Sprite achieves transparency by assigning each process a home node. A process' home node is the machine on which the process was created, unless the process was created by a migrated process. In which case, the child process' home node is the same as that of its parent. Whenever a process invokes a kernel call whose results are machine-dependent, the kernel call is forwarded to the process's home node (using the RPC mechanism) and executed there. This guarantees that the process produces the same results as if it were executing at home. To the outside world, the process still appears to be executing at home.

The implementation of process migration on Sprite differs from other implementations. One difference is the way in which the virtual memory of a process is transferred between machines. Another major difference is the way in which migration is made transparent to the migrated process. The simplest approach to process migration is:

- Stop the process from further execution.
- Transfer its state to the new machine, including registers and other execution state, virtual memory, and file access.
- Restart the process on its new machine, so that it may continue executing.

The virtual memory transfer is the dominant cost in migration, so various techniques have been applied in Sprite to reduce it.

2.2.3 AMOEBA [References: 9, 10, 11]

2.2.3.1 Overview

Amoeba is a distributed operating system resulting from joint efforts at the Vrije Universiteit and the Centre for Mathematics and Computer Science in Amsterdam, The Netherlands under the direction of Tanenbaum and Mullender. The basic problem with current networks of PCs and workstations is that they are not transparent. That is, the users are clearly conscious of the existence of multiple machines. In contrast, the kinds of systems envisioned for the 1990s appears to the users as a single, 1970s centralized timesharing system. Users of these types of systems are not aware of which processors their jobs are using and they are not aware of where their files are stored or how communication is taking place among the processes and machines. All the resource management is performed automatically by a distributed operating system. Few such systems have been designed, and even fewer have been implemented. Amoeba is a distributed operating system design that was designed to meet the vision of distributed systems in the 1990s.

The Amoeba architecture consists of four components: workstations, pool processors, specialized servers, and gateways. The workstations are intended to execute only processes that interact intensively with the user. The majority of applications do not usually interact much with the user and are run elsewhere. Amoeba uses a processor pool for providing most of the computing power.This pool typically consists of a large number of single-board computers, each with several megabytes of private memory and a network interface.When a user application runs, a number of processors can be allocated to run most of the application in parallel. When the user is finished, the processors are returned to the pool so they can be used for other work. Amoeba treats the number of processors dynamically, so new ones can be added as the user population grows. Faulty processors can be removed with a degree of fault tolerance. The third system component consists of specialized servers, machines which run dedicated processes that have unusual resource demands. Finally, there are gateways to other Amoeba systems that can only be accessed over wide area networks. The role of the gateway is to protect the local machines from the exact details of the protocols that must be used over the wide area links.

Amoeba is an object-based operating system that uses clients and servers. Client processes use remote procedure calls to send server processes requests to carry out operations on objects. Each object is both identified and protected by a capability. Capabilities contain the set of operations that the holder may carry out on the object. They contain enough redundancy and cryptographic protection in a secure environment to make it infeasible to infer a capability. Amoeba assumes that embedding capabilities in a huge address space provides adequate protection. Due to the cryptographic protection, capabilities are managed outside the kernel, by user processes themselves. The service port of a capability identifies the service managing the object while the object number specifies which object. The rights restrict which operations are permitted. The check field provides cryptographic protection to keep users from tampering with the other fields.

Objects are implemented by server processes that manage them. Capabilities are encoded with the object's server so that, given a capability, the system can easily find a server process that manages the corresponding object. The RPC system guarantees that requests and replies are delivered at most once and only to authorized processes. At the system level, objects are identified by their capabilities. However, at the level of user written programs, objects are named using a human-readable hierarchical naming scheme. The mapping is carried out by the Directory Service which maintains a mapping of ASCII path names onto capabilities. The Directory Server has mechanisms for performing atomic operations on arbitrary collections of name-to-capability mappings.

Currently, one file server is used, the Bullet Server. It is a simple file server that stores immutable files as contiguous byte strings both on disk and in its cache. The Amoeba kernel manages memory segments, supports processes containing multiple threads and handles interprocess communication. The process-management facilities allow remote process creation, debugging, checkpointing, and migration. All other services, (such as the directory service) are provided by user-level processes, in contrast to standard UNIX, which has a large monolithic kernel that provides these services. By putting as much as possible in user space, Amoeba has achieved a flexible system, and has done this without sacrificing performance. In the Amoeba design, concessions to existing operating systems and software were carefully avoided. Since it is useful to be able to run existing software on Amoeba, a UNIX emulation service, called Ajax was developed.

Amoeba' s design is that of a client performing operations on objects. Operations are implemented by making remote procedure calls. A client sends a request message to the service that manages the object. A server thread accepts the message, carries out the request, and sends a reply message back to the client. For reasons of performance and fault tolerance, frequently multiple server processes jointly manage a collection of objects of the same type to provide a service.

2.2.3.2 Remote Procedure Calls in Amoeba

The kernel provides three basic system calls to user processes. One sends a message to a server and then blocking until a reply is received. The second is used by servers to announce their willingness to accept messages addressed to a specific port. The third, also used by servers, sends replies. All communication in Amoeba is of the following form:

- A client sends a request to a server.
- The server accepts the request, does the work.
- The server sends the reply to the client.

A user- oriented interface has been built on top of these services to allow programming of objects and operations on these objects. A class corresponds to each type of object, and can be composed hierarchically. RPC messages consist of two parts, a header and a buffer. The header has a fixed format and contains addressing information (including the capability of the object to which the RPC refers), an operation code which selects the function to be called on the object, and some space for additional parameters. The buffer contains data.

Before a request for an operation on an object can be delivered to a server thread managing the object, the location of such a thread must he found. All capabilities contain a Service Port field, which identifies the service managing the object to which the capability refers. When a server thread responds to a request, it provides its service port to the kernel. The kernel records mapping information in an internal table. When a client thread initiates an operation, it is the kernel's job to find a server thread with an outstanding response that matches the port in the capability provided by the client. When Amoeba is run over a wide area network, with huge numbers of machines, each server wishing to export its service sends a special message to all the domains in which it wants its service known. In each such domain, a dummy process, called a "server agent" is created which responds and then waits until a request arrives. The request is then forwarded to the server for processing.

2.2.3.3 The Amoeba File System

Capabilities form the low-level naming mechanism of Amoeba, but are impractical. On Amoeba, a typical user has access to thousands of capabilities from the user' s own private objects, as well as the capabilities of public objects, such as the executables of commands, pool processors, data bases, and public files. Hierarchical directory structures are used in Amoeba to implement partially shared name spaces.

Directories are ordinary objects with a capability that is needed to use them. Members of a group can be given access to a directory by granting them the capability of the directory. Others can be denied access to the directory by not giving them the capability.

Amoeba provides a number of services that can be used to: look up an object name in a directory returning its capability; enter, and delete objects from directories. Since directories themselves are objects, a directory may contain capabilities for other directories, thus potentially allowing users to build an arbitrary graph structure.

The Amoeba Bullet Server is an unusual file server. It supports only three principal operations, read, create, and delete a files. When a file is created, the user normally provides all the data at once, creating the file and obtaining a capability for it. In most circumstances the user will immediately give the file a name and ask the Directory Server to enter the information in a directory. All files are immutable; once created they cannot be changed. Note that no write operation is supported. Since files cannot change, the Directory Server can replicate them at its leisure for redundancy without fear that a file may change in the meantime.

Since the final file size is known when a file is created, files are stored contiguously, both on the disk and in the Bullet Server's cache. The administrative information for a file is then reduced to its original size plus some ownership data. The complete administrative table is loaded into the Bullet Server's memory when it is booted. When a read operation is performed, the object number in the capability is used as an index into this table, and the file is read to the cache in a single disk operation. The Bullet file server can deliver large files from its cache, or consume large files into its cache at maximum RPC speeds.

Object names always refer to consistent objects; sets of names always refer to mutually consistent sets of objects. Atomic actions form a useful tool in Amoeba for achieving consistent updates to sets of objects. In Amoeba, the Directory Service takes care of atomic updates by allowing the mapping of arbitrary sets of names onto arbitrary sets of capabilities.

The Directory Server plays a crucial role in the system. Nearly every application depends on it for locating required capabilities. The Directory Service replicates its internal tables on multiple disks so that no single failure will halt operation. Techniques used are essentially the same as those used in fault-tolerant database systems. The Directory Server is not only relied on to be constantly available, it is also trusted to work correctly and never divulge a capability to an entity not entitled to see it. Security is an important aspect of the reliability of the directory service, since even a perfectly designed Directory Server may allow unauthorized users to catch glimpses of the data.

2.2.3.4 Process Management in Amoeba

Amoeba processes are allowed multiple threads of control. A process consists of a segmented virtual address space and one or more threads. Processes can be remotely created, destroyed, checkpointed, migrated and debugged. On a uniprocessor, threads run quasi-parallel; on a shared-memory multiprocessor, the number of threads that may run simultaneously is limited exclusively to the number of processors. Processes may not be split over more than one machine, and have explicit control over their address space. When a process' memory segment is mapped out, it remains in memory, although no longer as part of any process' address space. The unmapping operation returns a capability for the segment which can then be read and written like a file. Thus, one process can map a segment out and pass the capability to another process. The other process can then map the segment in again. If the processes are on different machines, the contents of the segment are copied. On the same machine, the kernel can use shortcuts for the same effect. A process is created by sending a process descriptor to a kernel within an execute process request.

A process descriptor consists of four parts. The host descriptor describes on which machine the process may run. A kernel that does not match the host descriptor will refuse to execute the process. The second part is the capability of the process, needed by every client that manipulates the process. Another part is the capability of a handler, a service that deals with process exit, exceptions, signals and other anomalies of the process.

The final process descriptor part, a memory map, has an entry for each segment in the address space of the new process. Each entry gives virtual address, segment length, segment mapping specifications (read only, read/write, or execute only), and the capability of a file or segment from which the new segment should be initialized. The thread map describes the initial state of each of the threads in the new process:

- processor status word

- program counter
- stack pointer
- stack base
- register values
- system call state.

This detailed tracking of thread state provides for the use of process descriptors for the representation of executable files, and for processes being migrated, debugged or checkpointed.

Amoeba processes have two states: running, or stunned. In the stunned state, a process exists, but does not execute instructions. A process being debugged is in the stunned state, for example. The low-level communication protocols in the operating system kernel respond with 'this-process-is-stunned' messages to attempts to communicate with the process. The sending kernel will keep trying to communicate until the process becomes running or is killed. Thus, communication with a process being interactively debugged continues to work.

A running process can be stunned by a stun request directed to it from the outside world by it owner, or by an unhandled exception. When a process becomes stunned, the kernel sends its state in a process descriptor to a handler whose identity is a capability which is part of the process' state. After examining the process descriptor, and possibly modifying it or the stunned process' memory, the handler can reply either with a resume or kill command. Debugging of processes is facilitated through this mechanism. The debugger takes on the role of the handler.

Migration is also facilitated through stunning. First, the candidate process is stunned; then, the handler gives the process descriptor to the new host. The new host fetches memory contents from the old host in a series of file read requests, starts the process and returns the capability of the new process to the handler. Finally, the handler returns a kill reply to the old host. Processes communicating with a process being migrated will receive 'process-is-stunned' replies to their attempts until the process on the old host is killed. Then they will get a 'process-not-here' reaction. After locating the process again, communication will resume with the process on the new host. Command interpreters can cache process descriptors of the programs they start. Kernels can cache code segments of the processes they run. Combined, these caching techniques make process start-up times in Amoeba very short.

Amoeba is a distributed operating system with a system interface quite different from that of the popular standard UNIX-based operating systems of today. To support the large number of existing

UNIX applications, Amoeba contains a UNIX emulator. This emulator provides an environment to port existing UNIX applications to Amoeba with only minor changes. The emulation services are not binary compatible with other UNIX operating systems. In spite of the design-independence from UNIX, it remarkably easy to port UNIX software to Amoeba. The programs that are hard to port are mostly those not needed in Amoeba anyway.

From a security perspective Amoeba possess some interesting characteristics. When a new service is being designed, the protection of its objects usually does not require any thought because the use of capabilities automatically provides enough of a protection mechanism. It gives a very uniform and decentralized object management mechanism.

2.2.4 PLAN 9 [References: 12]

Plan 9 is a general-purpose, multi-user, portable distributed operating system developed at AT&T Bell Laboratories. It is currently implemented on a variety of computers and networks. PLAN 9 provides an excellent development and application environment but lacks a number of features often found in other distributed systems, including:

- uniform distributed name space,
- process migration, light weight processes or threads,
- distributed file caching,
- personalized workstation environments, and
- X windows

Plan 9 grew out of a desire at Bell Labs to build a system that could profit from continuing improvements in personal machines with bitmap graphics, availability in medium- and high-speed net- works, and in emerging high-performance microprocessors.

The developers of Plan 9 recognized early in their efforts that a common approach being taken to develop distributed systems is to connect a group of workstations with a medium-speed network. This approach unfortunately has a number of failings. Because each workstation has private data, each must be administered separately making maintenance difficult to centralize. At the rate of current vendor efforts, the machines are replaced every couple of years to take advantage of technological improvements, rendering the hardware obsolete. Plan 9 is completely new system that includes a compiler, operating system, networking software, command interpreter, window system, and the terminal.

Plan 9 is architected along the lines of a service function. CPU servers concentrate computing power into large (not overloaded) multiprocessors, file servers provide repositories for storage, and terminals give each user of the system a dedicated computer with a bitmapped graphical screen and mouse on which to run a window system.

The components communicate by a single protocol, built above a reliable data transport layer offered by an appropriate network which defines each service as a rooted tree of files. Even for services not usually considered as files, the unified design permits some simplification. Each process has a local file name space that contains attachments to all services the process is using and thereby to the files in those services. One of the most important jobs of a terminal is to support its user's customized view of the entire system as represented by the services visible in the name space. To be used effectively, the system requires a CPU server, a file server, and a terminal. The CPU server and file server are large machines. The system's strength stems in part from economies of scale.

Several computers provide CPU services under Plan 9. The operating system provides a conventional view of processes, based on fork and exec system calls, and of files, mostly determined by the remote file server. Once a connection to the CPU server is established, the user may begin executing commands to a command interpreter in a conventional-looking environment. Support for a multiprocessor CPU server has several advantages. The most important is its ability to absorb load. The CPU server performs compilation, text processing, and other applications. It has no local storage. All of the CPU server's permanent files it accesses are provided by remote servers. Transient parts of the name space, such as the collected images of active processes or services provided by user processes, may reside locally but these disappear when the CPU server is rebooted.

The Plan 9 file servers hold all permanent files. The file server presents to its clients a file system rather than, say, an array of disks or blocks or files. The files are named by slash-separated components that label branches of a tree, and may be addressed for I/O at the byte level. The location of a file in the server is invisible to the client. The contents of recently-used files reside in RAM and are sent to the CPU server rapidly by DMA over a high-speed link, which is much faster than regular disk although not as fast as local memory. With the high-speed links, it is unnecessary for clients to cache data. Instead, the file server centralizes the caching for all its clients, avoiding the problems of distributed caches. The file server actually presents several file systems. One, the "main" system, is used as the file system for most clients. Other systems provide less generally-used data for private applications. A backup system is provided in Plan 9 by the same file server and the same mechanism as the original files so permissions in the backup system are identical to those in the main system; one cannot use the backup data to subvert security.

The standard terminal for Plan 9 is called Gnot and consists of a Bell Labs developed machine. The terminal's hardware is reminiscent of a diskless workstation: 4 or 8 megabytes of memory, a 25MHz 68020 processor, a 1024x1024 pixel display with two bits per pixel, a keyboard, and a mouse. It has no external storage and no expansion bus; it is a terminal, not a workstation. A 2 megabit per second packet-switched distribution network connects the terminals to the CPU and file servers. Although the bandwidth is low for applications such as compilation, it is more than adequate for the terminal's intended purpose: to provide a window system, that is, a multiplexed interface to the rest of Plan 9. Unlike a workstation, the Gnot does not handle compilation; that is done by the CPU server. The terminal runs a version of the CPU server's operating system, configured for a single, smaller processor with support for bitmap graphics, and uses that to run programs such as a window system and a text editor. Files are provided by the standard file server over the terminal's network connection. All Gnot terminal are equivalent; they have no private storage either locally or on the file server.

Plan 9 has a variety of networks that connect the components. CPU servers and file servers communicate over back-to-back DMA controllers, a practical solution exclusively for computer center or departmental computing resource. More distant machines are connected by traditional networks such as Ethernet. A terminal or CPU server may use a remote file server completely transparently except for performance considerations. Gnot terminals employ an inexpensive network that uses standard telephone wire and a single chip interface.

The name space is modifiable on a per-process level, although in practice the name space is assembled at log-in time and shared by all that user's processes. To log in to the system, a user instructs a terminal to connect to a file server. The terminal contacts the server, authenticates the user, and loads the operating system from the server. The server then reads a file, called the profile, in the user's personal directory. The profile contains commands that define which services are to be used by default and where in the local name space they are to be attached. The profile then typically starts the window system. Each window in the window system executes a command interpreter that may be used to run commands locally. Locally executed commands use file names interpreted in the name space assembled by the profile.

For computation-intensive applications such as compilation, the user executes a command that can automatically select a CPU server to run the application. The user receives a normal prompt from the command interpreter running on the CPU server in the remote CPU server's name space. The terminal exports a description of the name space to the CPU server, which then assembles an identical name space, so the customized view of the system assembled by the terminal is the same as that seen on the CPU server.

An example of a local service found in Plan 9 is the 'process file system', which permits examination and debugging of executing processes through a file-oriented interface. Plan 9 implements a unique process management system. The root of the process file system is conventionally attached to the directory /proc. After attachment, the directory /proc itself contains one subdirectory for each local process in the system, with the name equal to the numerical unique identifier of that process. Each subdirectory contains a set of files which implement the view of that process. The list of files includes:

- mem, which is the virtual memory of the process image
- ctl, which controls behavior of the processes. Messages sent to this file cause the process to stop, terminate, or resume execution.
- text, which is the file from which the program originated. This is typically used by a debugger to examine the symbol table of the target process.
- note, which is available to any process with suitable permissions to send it an asynchronous message for interprocess communication. The system also uses this file to send poisoned messages when a process misbehaves, such as dividing by zero.
- status, which is a fixed-format ASCII representation of the process status. It includes the name of the file executing within the process, the CPU time it has consumed, its current state, etc. The status file illustrates how heterogeneity and portability can be handled by a file server model for system functions.

In Plan 9, user programs, as well as specialized stand-alone servers, may provide file services. The window system is an example of such a program. One of Plan 9's most unusual aspects is that the window system is implemented as a user-level file server. The window system is a server that presents a file named /dev/cons to the client processes running in its windows. When a new window is made, the window system allocates a new /dev/cons file, puts it in a new name space for the new client, and begins a client process in that window. That process connects the standard input and output channels to /dev/cons using the normal file opening system call and executes a command interpreter. When the command interpreter prints a prompt, it will be written to /dev/cons and appear in the appropriate window.

It is useful to compare Plan 9's windowing system approach to other operating systems. Most operating systems provide a file like /dev/cons which is an alias for the terminal connected to a process. A process that opens the special file accesses its associated terminal without knowing the terminal's precise name. Since the alias is usually provided by special arrangement in the operating system, it can be difficult for a window system to guarantee that its client processes can access their window through this file.

In Plan 9, a set of processes in a window shares a name space and in particular /dev/cons, so by multiplexing /dev/cons and forcing all textual input and output to go through that file the window system can simulate the expected properties of the file. The window system serves several files, all conventionally attached to the directory of devices, /dev. These include cons, the port for ASCII I/O; mouse, a file that reports the position of the mouse; and bitblt, which may be written messages to execute bitmap graphics primitives. Much as the different cons files keep separate clients' output in separate windows, the mouse and bitblt files are implemented by the window system in a way that keeps the various clients independent. Since the window system has no bitmap graphics code all its graphics operations are executed by writing standard messages to a file and as a result the window system may be run on any machine that has /dev/bitblt in its name space, including the CPU server.

The cpu command connects from a terminal to a CPU server, using a full-duplex network connection. Through a setup process, the terminal and CPU server exchange information about the user and name space. Then the terminal-resident process becomes a user-level file server that makes the terminal's private files visible from the CPU server. The command interpreter then reads commands from, and prints results on, its file /dev/cons, which is connected through the terminal process to the appropriate window on the terminal.

Graphics programs such as bitmap editors also may be executed on the CPU server since their definition is entirely based on I/O on files 'served' by the terminal for the CPU server. The connection to the CPU server is transparent, permitting heterogeneity. The CPU server and the terminal may be different types of processors. However, there are two distinct problems: binary data and executable code. Binary data can be handled two ways: by making it not binary or by strictly defining the format of the data at the byte level. For files that are I/O intensive, such as /dev/bitblt, the overhead of an ASCII interface can be prohibitive. In Plan 9, these files accept a binary format in which the byte order is predefined. Programs that access the files use portable libraries that make no assumptions about the order. Thus /dev/bitblt is usable from any machine, not just the terminal. This principle is used throughout Plan 9. For instance, the format of the compilers' object files and libraries is similarly defined; object files are independent of the type of the CPU that compiled them. Plan 9 provides an adequate but non-elegant solution to the problem of different executables.

Unfortunately, Plan 9 does not directly address security issues. However, some of its features are interesting. For example, breaking the file server away from the CPU server enhances the possibilities for stronger security. Since the file server is a separate machine, which can only be accessed over the network by the standard protocol, it only serves files; it cannot run programs. Many security issues are resolved by the simple observation that the CPU server and file server communicate using a rigorously controlled interface through which it is impossible to gain special privileges.

Plan 9 was originally designed to support lightweight threads. However, with the ability to share local memory and with efficient process creation and switching, both of which are in Plan 9, the functionality of threads can be matched. Process migration was also absent from the original design of Plan 9.

3.0 Real-Time Operating Systems

3.1 General Issues

Real time applications present complex and demanding problems to application developers. Not only must the application developer deal with the same problems that exist for other high-performance, parallel and distributed application programs, they must also deal with real-time operating system primitives for task control, interprocess communication, and device operation.

In order to support the development of real-time applications, real-time operating systems must offer specialized support. Since the current and future target architectures of real-time applications vary widely in size and complexity real-time operating system kernels must vary in size and functionality. Therefore, real-time kernels must be highly configurable in size and functionality.

Complex real-time applications also consist of parallel application tasks of differing sizes. Real-time operating systems must offer support for multiple task sizes, parallelism and communication latencies. Tasks are time-critical and task deadlines may vary in semantics and laxity. As a result, tasks must be executed within application-specific time constraints. Since no single task scheduler is likely to satisfy all real-time applications, real-time operating systems must be configurable, even at the lowest operating system levels. It appears that a single model of task communication is inefficient for all real-time applications. Since tasks are heterogeneous in their behavior to single messages, real-time operating systems must support multiple models of time-critical task communication.

Perhaps the most important aspect of a real-time kernel is its predictability. If it is possible to demonstrate that system specifications are real (subject to assumptions regarding operating conditions), the real-time system is said to be predictable. There are at least five characteristics under which real-time systems can be categorized:

- granularity of deadlines and task laxities
- strictness of deadlines
- reliability requirements of the system
- size of system and degree of interaction among components, and
- characteristics of the operating environment

Granularity and laxity will determine the length of tasks and amount of time available to make scheduling decisions. For a system with low task granularities fast, close-to-optimal scheduling decisions are more important than making the best decision. Strictness of deadlines determines whether the real-time system can be called hard or soft. Strictness also determines the usefulness of continuing to perform a task after its deadline has past. Obviously, different approaches are needed for soft-deadline systems as opposed to hard-deadline ones. The reliability of a system refers to whether or not critical tasks can always meet their deadlines. While simple systems may require that every arriving task meet its deadline, a system operating in a non-deterministic environment may be such that it is sufficient to keep the deadlines of critical tasks and a portion of the non-critical tasks, depending on the system's state.

In this section of the report, we focus on several general issues currently undergoing active research, and then discuss several real-time operating systems that have been constructed at academic institutions or that are available commercially.

Research on real-time scheduling has experienced a major shift during the last few years, from static to dynamic (or on-line) real-time scheduling. Early research focussed on relatively small-scale or static real-time systems, where task execution times were estimated prior to task execution and where the resulting task schedules could be determined off-line. The resulting scheduling algorithms address both periodic and sporadic tasks. Periodic tasks typically arise from sensor data and control loops. Sporadic tasks arise from asynchronous events or operator actions.

An effective scheduling algorithm jointly schedules all periodic and sporadic tasks so that the timing requirements (such as deadlines, execution rates, etc.) for both sets of tasks are met. Most popular among static algorithms are the cyclic schedulers, and more recently, the Rate Monotonic (RM) algorithms. The popularity id due in part to the ease with which they are mapped to priority-based low-level task schedulers. The rate monotonic algorithm assigns fixed priorities to tasks with different execution rates, with the highest priority being assigned to the highest frequency tasks and lowest priority to the lowest frequency task. One of the problems with RM algorithms is their lack of support for dynamically changing periods. Their schedulable bound also is less than 100%. A second problem with RM scheduling is priority inversion, which arises when a high priority job must wait for a lower priority job to execute.

In contrast, the Earliest Deadline First (EDF) scheduling algorithm can be used for dynamic as well as static scheduling. This algorithm uses the deadline of a task as its priority. The task with the earliest deadline has the highest priority. Since priorities are dynamic, the periods of tasks can be changed at any time. A variant of ELD scheduling is Minimum-Laxity-First (MLF) scheduling. In this scheme, laxity is assigned to each task in the system, and minimum laxity tasks are executed earliest. Laxity measures the amount of time remaining before a task's deadline will pass if the task uses its allotted execution time. Essentially, laxity is a measure of the flexibility available for scheduling a task.

EDF is superior to RM in the sense that its schedulable bound is 100% for all tasks. However, a known problem with EDF is that there is no way to guarantee which tasks will fail in transient overload situations. This problem has lead to another variant called the Maximum-Urgency-First (MUF) algorithm. In this algorithm, each task is given an explicit description of urgency. This urgency is defined as a combination of two fixed priorities, and a dynamic priority, which is inversely proportional to the task's laxity. One of the fixed priorities, called task criticality, has precedence over the task's dynamic priority. The other fixed priority, called user priority, has lower precedence than the task's dynamic priority. The algorithm combines EDF scheduling with a synchronization scheme for access to shared resources.

3.1.1 Evaluation and Algorithm Improvements

A number of studies of both preemptive and non-preemptive scheduling algorithms in the context of real-time systems have been performed. Rate-monotonic and earliest deadline scheduling algorithms are optimal for static priority and dynamic priority scheduling algorithms, respectively. It is also known that an optimal scheduler cannot be found for multiprocessors unless a priori knowledge exists of the deadlines, computation times and arrival times of all the tasks. One study showed that even for a single processor, constructing a schedule with arbitrary arrival, computation and laxity requirements is a difficult problem. In a hard real-time environment, a task which expires while in the queue is discarded and not considered for service, whereas in a soft real time environment, such a task is retained in the queue and is still eligible for service.

The best algorithms for maximizing the value of completed tasks for multiprocessor systems takes into account the expected value of a task at completion. The expected value of a task is the probability that a task completes prior to its critical time or deadline.

3.2 Specific Issues Associated with Real-Time Operating Systems

3.2.1 Imprecise Computations

Recent work in real-time schedulers considers changes in the semantics of timing constraints to be used and enforced in actual systems. The premise being that there are some algorithms (e.g., iterative algorithms) that can return results at almost any time during their execution. The longer they run, the more precise their results. The result produced by a prematurely terminated process may be not as precise as desired, but it may still be acceptable, and therefore, can be used by the application.

3.2.2 Effects of Cycle-Stealing on Scheduling Algorithms

The effects of cycle stealing on scheduling algorithms in a hard real-time environment is also being studied. An I/O device can transfer data by direct memory access (DMA) and steal cycles from the processor, and therefore from the executing task. In a real-time system, cycle-stealing can create delays causing task deadlines at a low degree of processor utilization to be missed. Often, I/O devices are designed in such a way that FIFO is the only possible way to schedule I/O activity.

3.2.3 Schedulers for Multiprocessor and Distributed Real-Time Systems

More recent work in real-time scheduling addresses uniprocessor and multiprocessor systems, where scheduling must be performed on-line for both sporadic and periodic arrivals. One of the few on-line multiprocessor algorithms in the literature is an any fit algorithm which offers a distributed implementation. Global schedulers perform the assignment of tasks to processors in cooperation with processor-local schedulers which carry out deadline scheduling. This algorithm is similar to bidding algorithms developed for distributed systems.

3.2.4 Synchronization

Synchronization in hard real-time systems is a relatively new concern. The important problem that arises in this context is the effect of blocking caused by the need for synchronization tasks that require exclusive access to shared resources.

3.3 Analysis of Existing Real-Time Systems

3.3.1 Current Efforts

Research on real-time operating systems in the U.S. has been driven by three primary concerns: (1) support of the Ada language, (2) the predictable execution of embedded systems, and (3) dealing with the complexity of large-scale and dynamic real-time applications. Furthermore, recent research is focusing on providing a platform on which diverse real-time systems may be constructed. Commercial systems, on the other hand, have typically provided either Ada support, or some fixed set of primitives (i.e., micro-kernels) at the process level, or they have focussed on building real-time extensions to or alterations of Unix. The result has been the POSIX real-time standards for Unix, which require that the low-level scheduler in Unix be priority based and may be exchanged for a different scheduler. The DOD is also supporting real-time versions of the Mach operating system.

3.3.2 The Maruti Distributed Real-Time Operating System [References: 13, 14,

15]

The main focus of the Maruti project at the University of Maryland is to examine the constructs of future distributed, hard real-time, fault tolerant, secure operating systems. Maruti is an object-based system, with encapsulation of services. Objects consist of two main parts: a control part (or joint), which is an auxiliary data structure associated with every object; and a set of service access points (SAPs) which are entry points for the services offered by an object. Each joint maintains the object's information (such as computation time, protection and security information) and requirements (such as service and resource requirements).

Timing information, maintained in the object, is dynamic and includes temporal relations among objects. A calendar, a data structure ordered by time, contains the name of the services that will be executed and the timing information for each execution.

In Maruti, each application is described in terms of a computation graph, which is a rooted directed acyclic graph. The vertices represent services and the arcs depict timing and data precedence between two vertices. Objects communicate with one another by serial links. Such links perform range and type checking of the information. Objects that reside in different sites need agents as representatives on remote sites.

Maruti is organized in three distinct levels: the kernel, the supervisor, and the application level. The kernel is the minimum set of servers needed at execution time and consists of a set of core-resident objects that include:

- A dispatcher which is invoked upon the completion of a service or at the start of another service.
- A loader which loads objects into memory.
- A time server which provides the knowledge of time to executing objects.
- A communication server responsible for sending and receiving messages.
- A resource manipulator responsible for resource management.

Supervisor objects in Maruti prepare all future computations, ensuring their timely execution by pre-allocation of resources. The supervisor level objects in Maruti are:

- The allocator which extracts the resource requirement from a requests resource graph and allocates the required resources.

- The verifiers which verify resource usage and reservation.
- The binder responsible for connecting communication objects, as well as for verifying that their semantic relations are properly established.
- The login server providing a user interface to Maruti
- A name server responsible for bridging different name spaces and keeping track of machine locations and status.

The initial implementation of Maruti on a network of SUN Unix workstations was followed by a partial native kernel implementation on DECStations, and is now being replaced by an implementation based on Mach.

3.3.3 The ARTS Distributed Operating System [References: 16, 17]

ARTS is a distributed real time operating system developed in the ART (Advanced Real-time Technology) project at Carnegie Mellon University. The goal of the ARTS operating system is to provide users with a predictable, analyzable, and reliable distributed real-time computing environment, so that a system designer can analyze the system at the design stage and predict whether real-time tasks with various types of system and task interactions can meet their timing requirements.

The novel aspects of ARTS are its initial focus on distributed real-time applications, its integrated support of monitoring tools used for timing evaluation and display, and its later support of real-time communication protocols addressing live video transmission.

Objects in ARTS are much like those developed earlier in the CHAOS systems. They can be passive or active. An active object contains one or more user defined threads. In the active object, the designer of the object is responsible for providing concurrency control among co-executing operations. An object can be implemented using the C++ language with real-time extensions, called RTC++. Each operation of an object has an associated worst case execution time, called a "time fence" value and a time exception handling routine. When the operation is invoked from a real-time thread, the operation is executed if there is enough remaining computation time allocated to the calling thread to complete the operation. Otherwise, the invocation is aborted and an exception is raised.

ARTS also provides real-time threads to users. Each thread has an associated procedure name and a stack descriptor which specifies the size and address of the thread's stack. A real-time thread can be hard real-time or soft real-time. A hard real-time thread must complete its activities by its deadline time whereas the timeliness of soft real-time threads are less important. A real-time thread can be defined as a periodic or an aperiodic thread based on the nature of its activities.

ARTS provides Lock and Unlock primitives to delimit critical regions. When a thread wants to Lock an already locked variable, it is enqueued on a priority queue of threads. If the priority of the calling thread is higher than the priority of the thread which is in the critical region, the priority of the thread in the critical section is raised to that of the calling thread, thereby preventing priority inversion. When the thread leaves the critical region, its original priority is restored.

The ARTS kernel implements an Integrated Time-Driven Scheduler (ITDS). The ITDS scheduler provides an interface between the scheduling policies and the rest of the operating system. An object oriented approach is used to implement the scheduler with the policies embedded in the scheduler object. Each instantiation of the scheduler may have a different scheduling policy governing the behavior of the object, with only one instantiation being active at a given time.

In the ARTS system, an extended rate monotonic scheduling paradigm is used for communication scheduling. The system integrates message and processor scheduling with a uniform priority management policy. ARTS implements a communication structure which is intended to serve as a test-bed for new communication algorithms and protocols, as well as new real-time hardware. Protocols such as VMTP have been successfully implemented on the ARTS kernel. Furthermore, a Real-time Transfer Protocol (RTP) has been developed to explore real-time communication issues. RTP features prioritized messages and a time fence mechanism. As stated earlier, the most interesting contributions of ARTS are the results of process monitoring and the research on multimedia communication protocols.

3.3.4 Real-Time Mach [References: 24, 25]

The Real-Time Mach (RT-Mach) effort was initiated at CMU to develop a version of the Mach kernel for real-time systems. RT-Mach is a fallout of the ARTS development and primarily addresses the real-time aspects of threads, synchronization, interprocess communication (IPC), a real-time toolset for system design and analysis, and some other mechanisms to support greater predictability. There are also extensions to Mach for multimedia applications. These capabilities include extensions for real-time scheduling, user-mode device drivers, and a temporal paging system.

Traditionally, threads are defined for non-real-time activity. RT-Mach extends the model of a thread with real-time attributes. While the definition of a periodic thread is much the same as in current literature (a new instantiation of the thread being scheduled at intervals determined by the period), the definition of an aperiodic thread in RT-Mach includes a worst-case execution time, after which the aperiodic thread recurs. Both periodic and aperiodic threads can have soft or hard deadlines. The semantic importance of a thread is specified via a value function associated with the thread. There are primitives to create, terminate, kill and suspend a thread in addition to those for reading and setting the attributes of a thread. RT-Mach uses an ITDS scheduler, which was developed for AELTS, and extended for RT-Mach. Mach provides processor sets with run queues specific to processor sets. The ITDS scheduler extends this approach hy allowing five different policies (Rate Monotonic, Fixed Priority, Round Robin, Round Robin with transferrable Server and Round Robin with Sporadic Server) on each processor set in RT-Mach. RT-Mach allows various synchronization policies to be used with locks and conditions, which are well-known entities used for synchronization when programming with threads. Policies can be specified with individual locks and conditions.

The real-time IPC extensions in RT-Mach are real-time synchronization results applied for IPC. Thus, FIFO queuing order of messages, while acceptable for standard Mach, may cause unbounded delays in receiving high-priority messages in RT-Mach. Priority-based queueing solves this problem. In addition, primitives exist to propagate priorities from the sender of a message to the receiver. Mechanisms are available to inherit priorities from the sender to the receiver of a message. Further, since there may be multiple receivers in a server, a decision has to be made about which recipient thread should process an incoming message. Message buffers in RT-Mach are allocated before communication is initiated, which results in unpredictable buffer allocation delays. A receiver thread for a message must declare itself as the receiver of the particular port before communication is initiated.

The other effort at enhancing Mach for real-time and multimedia applications has a slightly different concept of real-time threads. This approach uses deadline-driven threads for devices where the media does not deteriorate if operations to the device complete before a deadline. Event-driven threads are used for those devices where the operation has to be initiated immediately after an event occurs. Such devices may have deadlines, in addition, and the response deteriorates as response time increases. The characterization of real-time threads includes a start time, a deadline, a worst-case execution time, and a 'weight' which specifies the relative importance of the thread. Event notifications are similar to user-level device management: at an interrupt, a handler performs a small routine in kernel mode on the device, an asynchronous system trap is posted and the preemptive-deadline scheduler is invoked, and the user-level operations follow.

3.3.5 CHAOS System [References: 18, 19, 20]

CHAOS is a a family of object-based real-time operating system kernels that address portability, extensibility, and customizability for low-level and subsystem-level operations. The family is extensible in that new abstractions and functionalities can be added easily and efficiently. Domain or target machine specific features may be implemented while preserving the kernel interface for existing programs. It also provides an environment for experimenting with and prototyping of new operating system constructs and policies.

CHAOS is customizable in that existing kernel abstractions and functions can be modified easily. This is useful because it facilitates changes to an operating system for uses with different target architectures or application domains. The CHAOS system is considered portable because its implementation is based on the now widely accepted Mach Cthreads standard. However, upwardly compatible modifications have been made to the Cthread interface in order to accommodate real-time applications.

Another issue addressed by the CHAOS researchers is the application-specific, on-line monitoring of running real-time programs. The purpose of such monitoring is to use monitor data to adapt running programs in performance and functionality to changing external execution environments.

The CHAOS system is portable to multiple platforms due to its use of real-time threads as a lower layer. While CHAOS is not intended for commercial use, offshoots of it are being used in commercial robotics applications.

3.3.6 The Harris CX/RT Operating System [References: 21, 22, 23]

CX/RT is a real-time version of the CX/UX operating system that provides a low-overhead real-time kernel, which is compatible with the CX/UX kernel. The real-time kernel is a multiprocessor multithreaded, preemptive kernel.

The CX/RT operating system consists of the following:

- the real-time kernel
- a frequency- based task scheduler and related utility, command, and library interfaces
- a performance monitor and related utility, command, and library interfaces
- a data recording facility and related utility, command, library, and real-time data recording interfaces
- additional features that are common to both the CX/RT and the CX/UX kernels. These additional features are: static priority scheduling, memory resident processes, data sharing at the symbolic level between C, FORTRAN, Ada, and assembly language processes, binary semaphores, asynchronous input/output, contiguous disk files, direct disk input/output,

memory mapping, real-time process synchronization tools, user-level interrupt routines, and interrupt daemons.

3.3.6.1 CX/RT Real-Time Kernel

The CX/RT kernel is an alternate kernel that has been designed specifically for real-time processing. Although compatible with the CX/UX kernel, the CX/RT kernel has been modified to reduce the amount of kernel overhead that is normally incurred. Modifications relate to system accounting functions, user buffer validations, and priority recalculation. The CX/UX and CX/RT kernels utilized on multiprocessor systems are multithreaded and preemptive. Multi-threading the kernel makes it possible for more than one thread of execution to be in the kernel at one time. The preemptive kernel enables a process that is executing in kernel mode to be forced to relinquish the CPU, permitting quick response to high-priority processes.

Both the CX/UX and the CX/RT kernels accommodate static priority scheduling. Processes scheduled within a certain range of priorities or under certain scheduling policies or circumstances do not have their priorities changed by the operating system in response to their run-time behavior. The resulting benefits are reduced kernel overhead and increased user control.

3.3.6.2 Overview of CX/RT

CX/RT utilizes a frequency-based scheduler (FBS). The FBS is a high resolution task synchronization mechanism that enables processes to run at specifiable frequencies. This mechanism does not replace the standard priority-based CX/RT scheduler. It depends upon the kernel scheduler to order process execution according to process priority. Convenient access to the major functions associated with frequency-based scheduling is provided by the real-time services utilities. Access is also provided through libraries of routines that can be called from application programs written in Ada, C, and FORTRAN 77.

The performance monitor monitors use of the CPU by processes that are scheduled on a frequency-based scheduler. Values obtained can help to determine whether redistribution of processes among processors for improved load balancing and processing efficiency is needed.

Data recording is a CX/RT mechanism which records the values of data objects during program execution. Data objects can be specified by using symbolic variable names in Ada and FORTRAN object programs or by using logical addresses. Values can be displayed on the terminal screen and modified from the keyboard, and they can be recorded to disk or tape. Access to data recording services is also provided through libraries of routines that can be called from application programs written in Ada, C and FORTRAN 77.

Paging and swapping often add an unpredictable amount of system overhead time to application programs. To eliminate performance losses due to paging and swapping, CX/RT has been designed to make certain portions of a process's virtual address space resident.Memory locking library routines allow the programmer to lock all or a portion of a process's virtual address space in physical memory. Other system calls allow locking of an application's I/O buffer in physical memory. The shmctl system call locks a shared memory region into memory. The plock system call locks a process' text region, its data and stack regions, or its text, data, and stack regions into memory. Locked regions are immune to paging or swapping. In contrast to traditional UNIX operating systems, pages that are not resident at the time of the call are immediately faulted into memory and locked.

CX/RT makes it possible for a process to communicate and share data with other processes or with memory-mapped external devices through use of the shared memory mechanism. This capability is provided for the FORTRAN, C, Ada, and assembly languages. CX/RT provide a means of quick process synchronization that is applicable to single processor and multiprocessor systems.

Based upon the concept of binary semaphores, this synchronization mechanism allows processes to ensure that no more than one process is executing a critical region of code. It ensures that a context switch is scheduled immediately when a process releases a semaphore for which a higher priority process is waiting. As a result, the higher priority process is scheduled to execute as soon as possible. This capability is accommodated by system calls that enable the programmer to create, initialize, and release a binary semaphore and C library routines that enable the programmer to lock and unlock and release a binary semaphore.

The ability to perform I/O operations asynchronously means that a process can set up for an I/O operation and return without blocking on I/O completion. CX/RT is designed to accommodate asynchronous I/O with the following system calls: aread, awrite, acancel, and await. These calls enable performance of asynchronous read and write operations, wait for completion of an asynchronous I/O operation, and cancel a pending asynchronous I/O operation. In addition, the programmer can test for completion of an asynchronous I/O operation and, upon completion of the operation, execute a user-defined routine.

CX/RT makes it possible to create a file of a specified size and allocate contiguous disk space to it. Using contiguous disk files is advantageous in real-time applications because it allows the programmer to read from and write to a file with a minimum number of seek operations. Allocation of contiguous disk space to a file is accommodated by the fcntl system call and by a command that has been developed for that purpose, mknodc.

CX/RT allows a user-level process to read and write directly between disk into its virtual address space, bypassing intermediate operating system buffering and increasing the speed with which disk I/O operations are performed. This capability is provided for both contiguous and noncontiguous disk files, but maximum benefit is obtained by using direct disk I/O with contiguous disk files. Direct disk I/O is accomplished via the open system call.

A set of real-time process synchronization tools has been developed to provide solutions to the problems associated with synchronizing cooperating processes' access to data in shared memory. This set includes tools for controlling a process's vulnerability to rescheduling, serializing processes' access to critical sections with busy-wait mutual exclusion mechanisms, and coordinating client-server interaction among processes. From these tools, a mechanism for providing sleepy-wait mutual exclusion with bounded priority inversion can be constructed.

The CX/RT operating system provides the support necessary to allow a user-level process to connect a routine to an interrupt vector corresponding to a interrupt generated by a selected device. When a process enables the connection to an interrupt vector, it blocks in kernel mode and no longer executes at normal program level. The process executes at interrupt level executing the user-level interrupt handling routine when the connected interrupt becomes active. Related operating system support includes the capability to raise and lower the interrupt priority level (IPL) from a user-level process, the capability to control edge-triggered interrupts from a user-level process, and the capability to dynamically allocate interrupt vectors from the kernel interrupt vector table.

Kernel daemons may be used instead of interrupts to handle (1) reception of IP (Internet Protocol) packets; (2) processing of interrupts from devices on I/O Controller, devices on asynchronous ports, and TTY ports that are opened in the standard CX/UX TTY mode; and (3) processing of entries in the callout queue.

3.3.6.3 User-Level Device Drivers

The operating system provides support for user-level device drivers. A user-level device driver consists of library routines that allow an application program to perform I/O and control operations for a particular device directly from user level without entering the kernel. User-level device drivers benefit both system responsiveness and program responsiveness. The CX/RT operating system also has support for real-time serial communications providing low-overhead I/O to simple serial devices.

3.3.6.4 Real-Time Signals

Signal management facilities include real-time signal extensions that are based on IEEE Draft Standard P1003.4. These extensions include definition of a range of real-time signal numbers, support for queuing of multiple occurrences of a particular signal, and support for specification of an application-defined value when a signal is generated.

3.3.6.5 Modifications to Support Real-time

This section provides an explanation of the changes that have been made to the system kernel to meet the requirements of real-time processing. Two types of changes have been made:

- An alternate low-overhead real-time kernel, which is called CX/RT, has been developed.
- Both the CX/UX and CX/RT kernels utilized on multiprocessor systems have been multithreaded and made preemptive.

The CX/RT kernel is a low-overhead executive which provides faster context switch times and reduced entry and exit times for system calls and interrupts. All of the scheduling priorities that can be associated with active processes on a CX/RT system are static, that is, they are not subject to adjustment by the kernel in response to their run-time behavior. To reduce overhead, five significant modifications have been made to the real-time kernel:

- The parameter and buffer validation functions routinely performed when invoking a system call have been removed.
- Process accounting information is not maintained.
- The priority adjustment code has been removed.
- The checks for read and write access to a device are not performed, and the last access time for a device is not updated.
- User profiling is not supported.

Although all modifications help to reduce overhead, removing the user buffer validation function alone has reduced system time by up to 5 percent on certain bench-marks. It is important to note, however, that removing this modification makes the system vulnerable; an incorrect program which specifies invalid addresses in a parameter list can cause unpredictable and potentially catastrophic results. For this reason, Harris strongly recommends using the CX/UX kernel for development purposes and reserving use of the CX/RT kernel for formal testing and production. As long as the software is bug free, this modification to the kernel poses no threat to the reliable functioning of the system. Furthermore, the virtual memory subsystem provides protection for most invalid specifications by limiting the effects to the process that has specified the invalid address. Such protection is not available in many real-time executives.

Removing maintenance of process accounting information from kernel functions results in a streamlined operating system which includes only the essential components. Removing the CX/UX priority adjustment code has two benefits: (1) priorities that are assigned to processes by the user remain local and (2) the overhead associated with calculating CPU usage and priority adjustment values is removed from the periodic system clock interrupts. The real-time kernel does not provide support for user profiling.

Multi-threading the kernel provides the ability for two or more processes running on separate processors to execute in the kernel simultaneously. Traditional UNIX operating systems do not permit preemption of a process executing in the kernel. The process to which the CPU is allocated maintains control of the CPU until it voluntarily relinquishes control by blocking or exiting, preventing quick response to high-priority processes.

Making the kernel preemptive makes it possible for a process executing in the kernel to be forced to relinquish the CPU involuntarily - to be preempted, so that a process with a higher priority that is ready to run can be scheduled for execution.

Making the kernel preemptive requires that provisions be made for preventing corruption of shared kernel data. Such provisions have been made by identifying the critical regions of code in the kernel, and by controlling access to them through the use of spin locks and semaphores. Only one thread of execution is permitted in a particular critical region at one time; however, two or more processes running on different CPUs are permitted to enter different critical regions simultaneously. When the kernel is made preemptive, the priority inversion problem arises. This problem is defined as follows: A low priority may be preempted while it is in a critical region of code. A high priority process that is attempting to enter the critical region blocks until the first process leaves. The first process, competing for the CPU at its low priority, hinders the progress of the high priority process.

The priority inversion problem appears to have been solved on CX/RT systems through use of the concept of time-slice passing. The kernel passes what would otherwise be the high priority process' time slice to the low priority process long enough to permit the latter to leave its critical region.The high priority process then resumes execution.

Multi-threading the kernel, making it preemptive, and using time-slice passing to solve the priority inversion problem have made it possible for Harris to achieve the following objectives that are beneficial to real-time processing:

- To increase utilization of CPUs in a multiprocessor system
- To decrease kernel preemption latency
- To ensure that a high priority process encounters little delay once it begins execution

3.3.7 Real-Time Network Communications

A relatively recent topic in real-time systems research concerns real-time communication protocols. Because of the temporal and spatial constraints that exist in real-time systems, continuous communication requires special resource management. In order to overcome such spatial and temporal constraints of continuous communication media, a few transport protocols such as ST-II Stream Protocol II), SRP (Session Reservation Protocol), XTP (Express Transport Protocol), VMTP (Versatile Message Transport Protocol) and fast lightweight transport protocols have been proposed. These protocols can be divided into two classes: reservation and non-reservation based protocols. The ST-II, and SRP protocols reserve system resources, such as, processor execution time, buffers, and network bandwidth before transmitting any data. A similar resource reservation model, a real-time channel, has been proposed for a wide area network environment. Such reservation of resources requires significant operating system support. On the other hand, VMTP and XTP transfer data on a best-effort basis and without any resource reservation. However, they do not guarantee on the end-to-end delay for a given session.

Some of the better research efforts in real-time network communications have been performed at UC Berkeley. Recent research from their laboratories has resulted in the construction of a real-time IP protocol, called RTIP. A Capacity-Based Session Reservation (CBSRP) protocol has been developed in order to provide guaranteed end-to-end delivery of data through resource reservation in a local area network environment. CBSRP differs from ST-II and SRP in that it is capable of dynamically changing quality of service session parameters.

4.0 Multilevel Secure Operating Systems

4.1 General Issues

Over the past few years, a number of MLS operating systems have been developed, and a few have received a formal assurance rating from the Government. The most popular versions of these systems are UNIX-based operating systems, primarily based on monolithic System V kernels. Most of those being offered by vendors are targeted to meet B1 levels of assurance; some efforts are underway to provide general purpose B2 and B3 trusted UNIX operating systems.

This section of the report focuses on general issues related to trusted UNIX-based operating systems, including threats to these systems. Then the capabilities of two such systems are discussed.

4.1.1 Introduction

The focus of the general discussion is on UNIX-based trusted operating systems, since these represent the vast majority of existing systems, and those under development or evaluation. Typically, to provide a trusted UNIX operating system, its commands, utilities and subsystems must be modified to produce the trusted system's Trusted Computing Base (TCB). The TCB is the set of protection mechanisms enforcing the trusted system's security policy. The changes to the trusted system result in a system that is designed to meet the security requirements of either the TCSEC or Orange Book, or the Security Requirements for System High and Compartmented Mode Workstation (CMWREQS).

Although the trusted system has extended the UNIX system to enforce additional security checks, the basic mechanisms of the system remain the same. Compatibility at the binary program interface and at the user interface typically are design criteria for a trusted system. All the benefits that have made the UNIX system so popular typically remain in the trusted system. The trust enhancements have been made to incur as small a reduction in performance and unexpected system behavior as possible. However, the additional security requirements do add overhead to the trusted system administration staff. Not only must this staff be familiar with the tasks involved in administering a trusted system, they must also be familiar with the trusted system mechanisms so that they can understand the implications of their actions.

A trusted system is one which employs sufficient hardware and software integrity measures to allow its use for simultaneously processing a range of sensitive or classified information. A trusted system can be trusted to perform correctly in two important ways:

- The trusted system's operational features operate correctly and satisfy the computing needs of the system's users.

- The trusted system's security features enforce the site's security policy and offer adequate protection from threats.

A security policy is a statement of the rules and practices that regulate how an organization manages, protects, and distributes sensitive information. The trusted system's security policy maintains full compatibility with existing UNIX security mechanisms while expanding the protection of user and system information. An organization carries out its security policy by running the trusted system as described in this manual and by adhering to the administrative and procedural guidelines defined for the system. Understanding the concept of a Trusted Computing Base (TCB) is important to understanding a trusted system. As mentioned earlier, the TCB is the set of protection mechanisms that enforces the trusted system's security policy. It includes all of the code that runs with hardware privilege (that is, the operating system or kernel) and all code running in processes that cooperate with the operating system to enforce the security policy. The trusted system's TCB consists of the following:

- A modified UNIX kernel: The kernel, also known as the operating system, runs in the privileged execution mode of the system's CPU. The Trusted system's kernel is isolated from the rest of the system because it runs in a separate execution domain, i.e., the processor's protected supervisor state.
- A Trusted X Window System: The system may provide a trusted graphical user interface, for example, CMWs. The Compartmented Mode Workstation provides modified versions of the X Window System and the Motif Window Manager to create a trusted windowing environment.
- Trusted commands and utilities The trusted system corrects, modifies, and adds to traditional UNIX software. A TCB is typically defined in terms of subjects and objects. The TCB oversees and monitors interactions between subjects (active entities such as processes) and objects (passive entities such as files, devices, and inter-process communication mechanisms). The trusted system provides at least partial protection for a UNIX system and its users against a variety of threats and system compromises.

The most important of these threats are summarized in Table 1-1 Potential System Threats.

- Data disclosure: The threat of disclosure occurs when a user gains access to information for which that user does not have a need-to- know or for which that user is not cleared. Need-to-know restrictions are enforced by the trusted system's discretionary access control policy, which states how users may, at their own discretion, allow their information to be accessed by other users. Clearance restrictions are enforced by the trusted system's mandatory access control policy, which regulates the ability of users to access information that has been classified at different sensitivity levels.

- Loss of data integrity: The threat of integrity loss occurs when user or system information is overwritten, either intentionally or inadvertently. Loss of data integrity may occur from hardware failures or software failures. When a loss of data integrity occurs, an opportunity is created for an unauthorized user to change information that affects the ability of the trusted system to function properly.
- Declassification of information: The threat of information declassification occurs when a user who is not cleared for a particular category of information is able to view that information. This can also lead to the threat of inadvertent mixing of classifications.
- Loss of TCB integrity: The TCB enforces the trusted system's security policy. Any loss of integrity of TCB programs and files, including the executable copies of those programs in memory, constitutes a compromise of the integrity of the TCB itself and may lead to incorrect enforcement of the security policy.
- Denial of service: To function usefully, the trusted system must respond to requests for service. One way to compromise the usefulness of a system is to cause it to fail in its ability to process work. When denial of service occurs, users lose the ability to access their information. Depending upon the method of attack, the threat of denial of service may accompany any of the other threats mentioned above.

In a trusted UNIX operating system, all subjects are processes. Processes are the active entities on the system that accomplish work. The objects that are typically implemented by the operating system are:

- Process: A process is an object when it is the target of a software signal or of a process debugging action.
- Files: These are the containers of data, unstructured streams of bytes that grow as data is appended to them.
- Directories: The file system is organized into a tree-structured hierarchy, with directories at the nodes. Directories contain references to files and other directories.
- Special Files: Character and block special files name hardware devices and other software objects. The operations defined for special files are those allowed by the device drivers that are invoked when these files are accessed.

- Named Pipes: Data written into a pipe is read in first-in first-out order. A named pipe appears as a file in a directory, and is specified by a pathname.

- Unnamed Pipes: An unnamed pipe also contains data that is accessed in first-in first-out manner. An unnamed pipe is private to the process that creates it and may only be accessed by processes that are created by the process after it creates the pipe.

- Pseudo ttys: A pseudo tty is a two-way connection between processes. There are two file system objects associated with each pseudo tty. A process that emulates a terminal opens a controlling pseudo tty. A process that expects a terminal interface can then open the corresponding slave pseudo tty. Reads and writes to either side of the pseudo tty affect the process "on the other side" of the pseudo tty.

- Sockets: A socket is a communication endpoint, on which special operations are defined. Sockets are typically bound to other sockets, either on the same machine or across a network, and establish a logical connection between the processes which reference them.

- Stream head: A stream is a set of bidirectional data structures that allows layered communication protocols to be implemented through modules with well- defined interfaces. The stream head is the trusted system abstraction presented at the programming interface that is the equivalent of a communication endpoint. A process sends and receives messages to other processes, either across a network or on the same machine, using system calls that address the Stream head.

- Message Queues: A message queue is a mailbox holding typed messages. Messages may be retrieved by type or in RFO order.

- Semaphore Sets: A semaphore set contains a list of one or more semaphores, each of which maintains a counter that is accessed through special operations. A process may wait or signal one or more semaphores in the set.

- Shared Memory Segments: A shared memory segment refers to a portion of memory that may be mapped into one or more process' address space.

The objects that are implemented by the Trusted X Window Server found on CMWs are listed below. These items are only applicable to X Window Server-based systems and do not apply to general purpose UNIX systems.

- Window: A window is an abstraction of a displayed region on the terminal screen. Processes make requests to the Trusted X Window System Server to manipulate the contents and attributes of windows.
- Pixmaps: A pixmap is a three-dimensional array of bits. A pixmap is normally thought of as a two-dimensional array of pixels, where each pixel stores an N-bit value. N is said to be the depth of the pixmap.
- Colormaps: A colormap consists of a set of entries defining color values. The colormap associated with a window is used to display the contents of the window. Each pixel value indexes the colormap to produce RGB values that drive the color guns of a monitor. Depending on hardware limitations, one or more colormaps may be installed at one time, so that windows associated with those maps display with correct colors.
- Properties: Windows may have associated properties that consist of a name, a type, a data format, and some data. Clients may use properties to share information such as resize hints, program names, and icon fonts with a window manager. The trusted system uses properties to store window sensitivity labels, information labels, and input information labels.
- Atom: An atom is a unique ID corresponding to a string name. Atoms are used to identify properties, types, and selections.
- . Local X Graphic Objects: These are X objects that are local and private to a client. They are fonts, cursors, and graphics context.
- Server Global Control Information: These are global X server objects, read by all clients, modifiable only by trusted clients.

4.1.2 Trusted Operating System Evaluation Criteria

Under the auspices of the National Computer Security Center (NCSC), the U.S. Government sets standards for trusted systems and performs evaluations of systems submitted by vendors. During the evaluation process, the NCSC subjects these systems to analysis and testing of both operational features and security features to ensure that they meet requirements summarized in the TCSEC or Orange Book. The Orange Book describes a graded classification of secure systems and specifies the criteria that distinguish these classes. Each class offers increasing levels of features and assurance that guard against system compromise.

Two types of requirements have been defined for each defined security class: features and assurances. These requirements must be addressed during the design, implementation, and testing of a trusted system. Features are visible functions that carry out the security policy of a system. Assurance is a task that the trusted system implementor must complete to guarantee to system accreditors that the system meets specifications and that the specifications match the specified security requirement and protect against the threats summarized above.

Requirements are imposed upon Compartmented Mode Workstations (CMWs) in addition to the requirements found in the Orange Book. These products must meet the requirements specified in the Security Requirements for System High and Compartmented Mode Workstations (CMWREQS), a superset of the Orange Book requirements for B1 systems. CMW requirements are developed and administered under the auspices of the Defense Intelligence Agency (DIA). Security requirements beyond the B1 class are imposed on CMWs because these products must have enough built-in security to operate in a windowed environment, not simply as terminals, and to be able to intercommunicate securely with other workstations and with larger computers.

Using the TCSEC and interpretations of the TCSEC, the NCSC evaluates a system. Based on the ult of the evaluation, NCSC awards an overall rating showing the degree of trust that can be placed in the evaluated system.

Most of the trusted operating systems available today have been designed and implemented to exceed the Trusted Computer System Evaluation Criteria (TCSEC) requirements for the B1 class or to meet the requirements for a CMW as defined by the CMWREQS.

4.1.3 Trusted Operating System Security Requirements

This section summarizes the security requirements that a trusted operating system implementation must address.

Accountability -- Identification and Authorization: individuals must identify themselves to the trusted system, and the system must be able to authenticate users' identities.

Accountability -- Audit: Users must be accountable for their actions. The trusted system records each security-related event and the user who caused that event to occur, and places the information in an audit trail.

Object reuse: When an object is initially assigned, allocated, or reallocated to a subject, that Object must not contain any data that the subject is not authorized to access.

Discretionary access control (DAC): An owner of an object containing data must be able to allow or deny access to that object based on a need-to-know policy.

Sensitivity labels: The trusted system must maintain a sensitively label associated with each subject and object. This label is used in making access control decisions.

Information labels: The trusted system must maintain an information label associated with each subject and object. This label represents the most sensitive level of information that is contained in the subject or object. It also encodes any code words, dissemination and control markings, and handling caveats that may be associated with the information Mandatory access control (MAC) The trusted system must determine whether a particular subject has authorization to access a particular object based on the sensitivity labels of the subject and the object.

- Trusted path: The trusted system must provide a direct and distinct communication path between itself and system users. This path must be used for security-critical actions.
- System architecture: The trusted system hardware and software must be structured to isolate protected system resources via such mechanisms as separate execution domains and address spaces.
- System integrity: The trusted system must periodically validate the correct operation of the system and must ensure that the correct system is started by an authenticated individual.
- Trusted facility management: The trusted system must support distinct administrative roles, *each of which may be assigned to one or more users.*
- Trusted recovery: The trusted system must provide a mechanism for recovering securely from system failure.
- Trusted distribution: The trusted system must support an automated system control and distribution facility that ensures the integrity of newly generated systems.

4.1.3.1 Accountability -- Identification and Authentication

Trusted system identification and authentication (I&A) requirements state that the trusted system must associate a user identity with a subject (identification) and prove that the user is who he or she claims to be by checking against criteria that correctly establish a user's identity (authentication). I&A is part of a secure system's requirement to hold users accountable for their actions on the trusted system. In traditional UNIX systems, the user logs into a system by entering a user name and a password. The system searches the password database (/etc/passwd) for the user name. If the system finds the user name, it authenticates the user by comparing the entered password to the encrypted version of the password in the user's password database entry. The UNIX system enforces various rules concerning the characteristics of the password and the ability to change it but these rules have proved insufficient to guard systems against penetration.

The Password Management Guideline published by the U.S. Department of Defense presents a set of recommended practices for the design, implementation, and use of password-based user authentication mechanisms. The trusted system significantly extends I&A features in accordance with the Green Book guidelines.

For example, the trusted system must provide additional rules that enforce the types of passwords that can be used, must provide new procedures for generating and changing passwords, must change the location and protection of certain parts of the password database, must give the administrator greater control over user actions. The administrator has the responsibility for maintaining the authentication information that the trusted system stores for each user. Tasks include creating new user accounts and changing account information as users enter, leave, gain trust, or lose trust on the system. An especially important trust enhancement expands the concept of user identity. Each process in a traditional UNIX system has a real and effective user ID as well as a real and effective group ID and a supplementary group list. In traditional UNIX systems, a process with the effective user ID set to the root account can set these identifiers for any user. The trusted system adds a separate identifier called the login user ID (UID). The UID is an indelible stamp on every process associated with a user. The UID identities the user who is responsible for the process in session. After a process has an UID associated with it, the UID cannot be changed.

4.1.3.2 Accountability -- Audit

Trusted system requirements state that an audit capability must be supported. In a trusted system, all users are accountable for their actions. An action is accountable if it can be traced to a particular user. Traditional UNIX systems lack good accountability because some actions cannot be traced to any user. For example, pseudo-user accounts such as lp typically run anonymously. Their actions can be discovered only by changes to system information In addition, traditional UNIX systems keep a limited record of system actions. The accounting subsystem writes only a single accounting record on completion of each user process.

In a trusted UNIX system, each security-related action performed by a user is audited. All security-related events are recorded in an audit trail that can be examined to detect attempts at system penetration or other abuses of the trusted system's security policy. The audit trail provides an extensive analysis of the history of system changes. It can potentially record every change of subject, object, and system characteristics and can be used to identify the users responsible for system changes.

4.1.3.3 Object Reuse

Trusted system object reuse requirements state that when an object containing data is assigned, allocated, or reallocated, the trusted system must ensure that the object does not contain data left from the previous use of that object. In a system that does not control object reuse, information can be disclosed inadvertently when storage is released by one user and then made available to another user, bypassing the usual access checks. The trusted system must use UNIX system and X Window System mechanisms to clear newly allocated disk blocks, memory pages, and Window System objects. It extends the mechanisms of the UNIX system's authentication features to clear buffers containing clear text passwords immediately after these passwords are encrypted. Although this is unnecessary because a process' address space is protected, the password is the only basis for authentication, so all buffers containing clear text passwords are cleared immediately after use for good measure.

4.1.3.4 Discretionary Access Control

The trusted system's discretionary access control (DAC) requirements state the need to control access to information on a discretionary basis. With DAC, owners of objects containing data can allow and deny access to objects at their own discretion, according to a user controlled need-to-know policy. Users of traditional UNIX systems are accustomed to enforcing object protection through a relationship between the user and group(s) of a process and the owner, group, and mode bits of the object. The owner sets permissions on files that allow different accesses to the owner of the object, the group of the object, and the rest of the user community. These *protection attributes* are granted at the discretion of the owner. These attributes can be changed by the owner to be more restrictive (controlled access) or more permissive (open access).

Beginning at the C2 class of trust, the Orange Book requires that the discretionary controls on an object be "capable of including or excluding access to the granularity of a single user." For example, an owner of a file must be able to specify the name of one specific user who is authorized to access the file or to stipulate that all users except a particular user are authorized to access the file.

Although this requirement can theoretically be met using UNIX group mechanisms, it is very cumbersome when applied to a large community with many individual access needs. In addition to the UNIX owner, group, and mode word, trusted systems use access control lists (ACLs) to enforce discretionary access controls. An ACL for a file contains entries that specify permissions for each user or group allowed or denied access to the file. ACLs allow the owner of an object to maintain precise control, on an object-by-object basis, over who can access the object. Discretionary access controls are primarily an individual user's responsibility, rather than an administrator's.

4.1.3.5 Sensitivity Labels

Trusted Systems mandatory access control (MAC) requirements state that a trusted system must be able to maintain a sensitivity label with each subject and storage object. The trusted system must be trusted to store classified information and to guard it from disclosure. Meeting sensitivity label requirements ensures that:

- The sensitivity label applied to the object correctly reflects the object's sensitivity.
- The sensitivity label applied to the subject dictates that subject's rights to access the data stored in objects.

The sensitivity label consists of a combination of two components: a classification, and a set of compartments. The classification is one of a hierarchical set of classifications defined by the administrator. In a military environment, these classifications might be TOP SECRET, SECRET, CONFIDENTIAL, and UNCLASSIFIED. The set of compartments is a list of nonhierarchical categories. When a storage object is exported to printed paper or to magnetic media, the object's sensitivity label must accompany the object to the device, either externally on the device or internally with the information.

Data must be appropriately labeled so that users who are not cleared for the information may not access it. The data must be handled according to the procedures consistent with the data's sensitivity. It is crucial that the trusted system not allow data to flow from a higher labeled object (that is, more sensitive) to a lower labeled one. Such a flow of data would allow a user who is not cleared for the information to access it. The classifications and compartments defined at a particular site reflect the organization's policies and procedures. The trusted system enforces the information flow restrictions which implement the organization's information protection needs.

4.1.3.6 Information Labels

The trusted system's information label requirements state that every subject and object in the system must have an associated information label. Information labels are similar to the MAC sensitivity labels described in the previous section. Like sensitivity labels, information labels consist of a hierarchical classification and a set of nonhierarchical categories or compartments. Unlike sensitivity labels, which permit access control decisions to be made between subjects and objects, information labels do not contribute to the trusted system's access control decision. Information labels are used for two purposes. First, they represent the high water mark of the information contained in a subject or object. Second, they contain markings that encode any code words, dissemination and control markings, and handling caveats (for example, EYES ONLY, NO FOREIGNERS) that may be associated with the information. While a sensitivity label typically remains static for the life of a subject or object, the information label will be adjusted automatically as the information content of the subject or object changes.

In addition to associating information labels with processes, files and other objects, the trusted system provides an information label for each window system object. This label accurately represents the sensitivity of the information displayed in the window. The system also provides an input information label for each input action (for example, a key stroke) performed by a user.

4.1.3.7 Mandatory Access Control (MAC)

Trusted systems require mandatory access rules between labeled subjects (users and processes) and labeled objects (such as data). Mandatory access control rules are enforced by the trusted system. Their enforcement is not at the discretion of any user. MAC is structured as follows:

- To read an object, the subject's classification must be greater than or equal to the object's. The subject's set of compartments must include the object's. A process can read lower sensitivity data The data only raises in classification when it transfers from the contents of an object with a lower sensitivity label to the address space of a process with a higher label.
- To write an object, the object's classification must be equal to the subject's, and the object's compartment set must be equal to the subject's. The trusted system requires that subject and object labels are equal write data into the object. Most systems, however, provide a defined privilege that allows writing to objects with higher sensitivity labels.

In the area of mandatory access control, the administrator is responsible for setting the highest sensitivity level at which a user may create a session on the system. This level reflects the user's clearance to sensitive data The administrator is also responsible, under certain controlled circumstances, for changing object sensitivity levels, for example, when the system is corrupted or w! an object has inadvertently been over classified.

4.1.3.8 Privilege Mechanisms

Rather than basing all of its privilege checks on whether the process is associated with effective user ID 0 (root), as in traditional UNIX systems, trusted operating systems typically implement a more discrete privilege scheme. The trusted system defines a set of operational rights that may be individually held by processes. All places in the operating system that previously had a superuser check now check for the possession of one or more of the defined privileges.

Typically, the trusted system implements a superuser compatibility mode which allows an account or program to run with superuser semantics. When running in this mode, a process with user ID 0 has most of the defined privileges. The trusted system's privilege mechanism allows trusted programs to individually raise and lower privileges to support privilege bracketing around system calls which require privilege. Trusted programs are designated as those programs whose executable files have privilege sets that contain the privileges that program is trusted to invoke.

Privileges are enforced at the operating system interface level and affect the actions of programs. The trust associated with users is designated by assigning them command authorizations. In some systems, command authorizations are defined to control access to programs or program subfunctions. Rather than using discretionary identity to designate whether a user has access to an administrative or otherwise authorized function, the system uses command authorizations.

A user can perform any authorized function by invoking the appropriate command instead of changing discretionary identity. In traditional UNIX systems, administrators shared the root password in order to use *su* to perform administrative tasks. Command authorizations allow users to assume administrative roles without password sharing.

4.1.3.9 Trusted Path

The trusted path is a direct and unmistakably distinct communication path between the TCB and system users. This path is used whenever a connection that is guaranteed to be secure is required between a user and the TCB. Such a connection is required for security-critical actions such as changing passwords and specifying sensitivity and information labels to the trusted system. The use of the trusted path ensures that a user cannot be tricked into communicating with a process that pretends to be the trusted software. This type of trickery is sometimes known as spoofing.

Trusted systems typically implement a trusted path mechanism by reserving a portion of the screen, a portion of each displayed window, and a configurable key sequence for trusted path invocation.

4.1.3.10 Identification of User Terminal

Trusted system user terminal identification requirements state that the trusted system must provide a mechanism for positively identifying all user terminals and other user-accessible devices (for example, tape drives and printers) before allowing them to access system resources.

4.1.3.11 System Architecture

System architecture requirements state the need to isolate the system resources to be protected so that they are subject to access control and auditing requirements. To ensure protection, security functions are implemented in the operating system, not in applications. This requirement is met with both hardware and software features. Hardware features such as separate execution domains and segmentation are used to separate protection-critical elements. Software features include the structuring of the operating system into independent modules, a well-defined user interface to the operating system, and the use of distinct address spaces to maintain process isolation.

4.1.3.12 Trusted Facility Management

Trusted facility management requirements state that trusted systems must support three distinct administrator roles. Entry to and exit from a role must be a distinct and auditable event, and any user with the proper authorization can assume any role regardless of his login. The administrator roles are: Information System Security Officer (ISSO), System Administrator, and Operator. Each role is responsible for specific administrative functions and has certain operational rights.

4.1.3.13 Trusted Recovery

Trusted recovery requirements state that recovery without protection compromise must be achievable after a system failure. The trusted system maintains the integrity of sensitivity levels and information labels across system crashes.

4.1.3.14 Trusted Distribution

Trusted system trusted distribution requirements state that a system must provide a trusted automated system control and distribution facility. This facility maintains the integrity between the master data describing the current version of the system and the on-site master copy of the code for the current version. It also provides assurance that any updates distributed to a customer are exactly as specified by the master copies.

The administrator of a trusted UNIX system is responsible for traditional UNIX administrative functions as well as for many additional security functions. The following list shows some of the key responsibilities of administrators in any UNIX system:

- Installing the system
- Configuring file systems
- Establishing user accounts
- Backing up and restoring the system's data
- Maintaining a variety of subsystems
- Installing applications
- Monitoring and tuning system performance
- Responding to events that require administrator attention.

In a trusted system, additional administrator responsibilities include the following:

- Setting up security databases
- Maintaining the additional security parameters of users' authentication profiles
- Monitoring the security and integrity of the trusted system
- Auditing security-related events and maintaining the trusted system's audit functions
- Performing trusted recovery
- Performing miscellaneous administrative tasks associated with protected subsystems
- Performing functions that are reserved to authorized users on behalf of unauthorized users (for example, changing DAC protection on files when users make mistakes)

An important difference between traditional UNIX and the trusted systems is in the area of system administration. The administrator must install, set up, and maintain the system so that it will protect the information stored in the system. Administrator actions are crucial to maintaining a trusted system. An effective administrator must understand the system's security policy, how it is controlled by the information entered in the trusted system's security databases, and how any changes made in these databases may affect user and administrator actions. The administrator must also be aware of the sensitivity of the information being protected at a site, the degree to which users are aware of and willing and able to cooperate with the trusted system's security policy, and the threat of penetration or misuse from insiders and outsiders. In a traditional UNIX system, the root account is typically responsible for handling all user administration, accounting, and operational tasks. In trusted systems, administrative tasks are split among three distinct logical roles (ISSO, System Administrator, and Operator), each of which is responsible for one particular part of the total administration of the system. Any user may be given the authorization to act in one or more of these roles.

4.2 Analysis of Existing MLS Operating Systems

4.2.1 SecureWare Compartmented Mode Workstation (CMW+) [References: 26, 27]

4.2.1.1 Overview

CMW+ is SecureWare's multi-user, multilevel, networked trusted compartmented mode workstation. Open Desktop (ODT) is a product of the Santa Cruz Operation (SCO), and is a version of SCO's UNIX System V/386 Release 3.2. Latest releases also includes SecureWare's MAXSIX technology. MAXSIX is a trusted network architecture that allows computers with differing security policies to interoperate across a network such that the security policies of each system will be enforced. The SecureWare security software and the MAXSIX trusted networking software have been licensed by a number of secure operating system vendors for inclusion into their operating system variants of UNIX.

SecureWare's CMW+ product modifies the standard UNIX operating system (in this case, SCO's ODT) and its commands, utilities, and subsystems to produce a trusted operating system that includes a Trusted Computing Base (TCB). The TCB is the set of protection mechanisms that enforce the system's security policy. It includes all of the code that runs with hardware privilege (that is, the operating system or kernel) and all of the code running in processes that cooperate with the operating system to enforce security. A TCB is typically defined in terms of subjects and objects. The TCB oversees and monitors interactions between subjects (active entities such as processes) and objects (passive entities such as files, devices, and interprocess communication mechanisms).

The trusted system consists of the following:

- A modified UNIX kernel, also known as the operating system, that runs in the privileged execution mode of the system's central processing unit (CPU).
- A Trusted X Window System that includes modified versions of the X Window System and the Motif Window Manager to create a trusted windowing environment. The user may run several X Window sessions simultaneously at different security levels, while maintaining a strict separation of compartmented data. Information about the sensitivity of the data contained in each window is displayed in the border surrounding that window session.
- Trusted UNIX commands and utilities that are security relevant have been modified to enforce the CMW Least Privilege Requirements. In addition, several new commands have been added to help administer a trusted system.
- Trusted Networking components that include the MAXSIX trusted network, a set of extensions to the traditional communication protocols and the Interprocess Communication (IPC) mechanisms that access them. The trusted network supplies trusted communication services to the applications that use it.

4.2.1.2 Trusted CMW+ System Features

The remainder of this section discusses the main features of the Secureware trusted system.

The Secureware trusted operating system provides support for system identification and authentication (I&A) requirements that state that the trusted system must associate a user identity with a subject (identification) and prove that the user is who he or she claims to be by checking against criteria that correctly establish a user's identity (authentication). I&A is part of a trusted system's requirement to hold users accountable for their actions on the trusted system. As such, the trusted system has additional rules that enforce the types of passwords that can be used, provides new procedures for generating and changing passwords, and changes the location and protection of certain parts of the password database.

The Secureware trusted operating system provides support for identification of individual user requirements that state that an audit capability must be supported. In a trusted system, all users are accountable for their actions. An action is accountable if it can be traced to a particular user. Traditional UNIX systems lack good accountability because some actions cannot be traced to any user. For example, pseudo-user accounts such as *lp* typically run anonymously. Their actions can be discovered only by changes to system information. In addition, traditional UNIX systems keep a limited record of system actions. The accounting subsystem writes only a single accounting record on completion of each user process.

The Secureware trusted operating system provides support for object reuse requirements that state that when an object containing data is assigned, allocated, or reallocated, the trusted system must ensure that the object does not contain data left from the previous use of that object which the new subject (that is, the user) is not authorized to access.

The Secureware trusted operating system provides support for discretionary access control (DAC) requirements that state the need to control access to information on a discretionary basis. With DAC, owners of objects containing data can allow and deny access to objects at their own discretion, according to a user-controlled need-to- know policy.

The Secureware trusted operating system provides support for MAC requirements that state the need for access rules between labeled subjects and labeled objects. The trusted system must be trusted to store classified information and to guard it from disclosure. Mandatory access control rules are mandated by the trusted system. The Secureware trusted operating system provides support for privilege mechanism rather than basing all of its privilege checks on whether the process is associated with effective user ID O (root). As in traditional UNIX systems, the trusted operating system implements a discrete privilege scheme. The trusted system defines a set of operational rights that may be individually held by processes. All functions in the operating system that previously had a check for superuser now check for the possession of one or more of the defined privileges.

The Secureware trusted operating system provides support for trusted recovery requirements that state that recovery without protection compromise must be achievable after a system failure. The trusted system maintains the integrity of sensitivity and information labels across system failures.

The Secureware trusted operating system provides support for trusted facility management requirements that state that the trusted system must support distinct administrator roles. Entry to and exit from a role must be a distinct and auditable event. The administrator roles are:

- Information System Security Officer (ISSO)
- System Administrator
- Operator
- Network Security Officer (NSO)

Each role is responsible for specific administrative functions and has certain operational rights. The role being played by the user is communicated through command authorizations held by that user.

The Information System Security Officer OSSO) is primarily responsible for managing security-related mechanisms. The ISSO controls the way that users log in and identify themselves to the trusted system. The ISSO must cooperate with the System Administrator when performing security-related tasks. The system's design and segregation of duties often requires that each perform a separate part of a total task (for example, account creation). Specific ISSO responsibilities include:

- The ISSO assigns system-wide defaults.
- The ISSO modifies non ISSO user accounts.
- The ISSO performs device assignment.
- The ISSO audits system activity.
- The ISSO ensures system integrity.

To perform ISSO functions you must have the *isso* command authorization.

The System Administrator is primarily responsible for non-label-related asPects of system administration, including account creation and disabling, and for ensuring the internal integrity of the system software and network systems. Specific System Administrator responsibilities include:

- The System Administrator creates user accounts.

- The System Administrator creates groups.
- The System Administrator modifies ISSO accounts.
- The System Administrator creates file systems.
- The System Administrator performs trusted recovery.

To perform System Administrator functions you must have the *sysadmin* command authorization.

The Operator is primarily responsible for ensuring that day-to-day hardware and software operations are performed in trusted fashion. Specific Operator responsibilities include:

- The Operator administers line printers.
- The Operator backs up and restores files to recover from user errors and system failures

To perform Operator functions you must have the *operator* command authorization.

The NSO performs the following MAXSIX network operation tasks:

- The NSO sets up and maintains the MAXSIX networking databases
- The NSO fine tunes MAXSIX configuration parameters
- The NSO distinguishes MAXSIX-controlled operations from non-secure network operations
- The NSO troubleshoots network problems

To perform NSO functions you must have the *nso* command authorization.

4.2.1.3 Command Authorizations

The ISSO assigns command authorizations to individual user authentication profiles to designate how much trust each user has on the system. Trusted programs check whether the user has a required command authorization before performing an operation that is restricted to authorized users. The Secureware system implements a privilege model that distinguishes between privileges and authorizations. Privileges are inherently associated with programs and are necessary for a process to invoke sensitive system calls or trusted code that can compromise the security of the system if not used correctly. From the point for view of trusted applications, privileges are further broken down into potential and effective sets. The potential set contains all the privileges that a given program might need to function and is trusted to use. The potential set is attached to the executable program file by the administrator during installation of the software. The effective set refers to those privileges that trusted system calls actually recognize. trusted kernel routines use privilege bracketing to add and remove potential privileges to the effective privilege set during execution.

While command authorizations are inherently associated with programs, command authorizations are associated with individual users. They denote the security sensitive actions a user can take. Privileges are checked only by system calls. Authorizations are checked only by trusted programs, sometimes before a system call to which a corresponding privilege applies. For example, a trusted program might check the owner authorization before modifying some attribute of a file on behalf of a user. If the authorization is present, the program uses the owner privilege to perform a system call which allows modifying some attributes of files owned by other users. Secureware CMW+ includes a number of privileges to maintain compatibility with traditional UNIX systems, such as *execsuid*.

4.2.1.4 Multilevel Directories

Secureware CMW+ provides limited support for multilevel directories. This functionality is primarily intended to provide backward compatibility to traditional public directories such as /tmp. Secureware provides a specific privilege, *multileveldir*, that permits the traversal of a multilevel directory from a program. In Secureware, multilevel directories are implemented as a tree structure beneath a parent directory, such as /tmp. For each sensitivity level, a separate subdirectory is created when a file is created. The subdirectory sensitivity label matches that of the created file. These child directories are called diversion directories. A given diversion directory is shared only by unprivileged processes executing at the same sensitivity level as the directory. The need for multilevel directories is obvious. Many UNIX programs rely on common directories in which to store temporary or working files. Unfortunately, the current file system implementations, there is no MAC label that can be applied to a single directory that would permit it to be used by programs executing at different sensitivity levels. The Secureware multilevel directory overcomes this problem by automatically creating diversion directories for executing programs.

Secureware provides a trusted Trusted Window System which provides the following hardware and software capabilities:

- An X Window System graphics environment, including a trusted version of X and a trusted version of the Motif Window Manager
- A window-based user interface that allows you to interact with the trusted system via a trusted path mechanism

4.2.1.5 MAXSIX Trusted Network

MAXSIX is the trusted network component of the Secureware CMW+ system. MAXSIX adds extensions to the standard TCP/IP protocol stack to augment the security capabilities of standard UNIX networking. It allows a system to control data flow through its network subsystem as it would regulate data flow between local subjects and objects. Such data might represent a request to run a local program for a remote process -- such as the BSD *rsh* command -- or it might represent the contents of a file transferred as the result of running a file transfer program, such as *ftp*.

The MAXSIX security additions fall into two layers: the Session Management layer, and the Network layer. MAXSIX services are accessed by trusted applications through the MAXSIX Application Programming Interface (API). The MAXSIX API provides a standard interface to the underlying network security services. These services include authentication, data secrecy, data integrity and attribute transport. No security features are actually implemented within the library. Rather, the library acts as an interface to the underlying session management layer.

The term session management is a bit misleading. It does not have any relation to the session layer defined within the ISO network model, but refers instead to the protocol that defines the security features required on a per-session basis, as first identified in the DODIIS Network Security for Information Exchange (DNSIX) specifications. (DODIIS is the Department of Defense Intelligence Information System.) While the actual services depend on the session management protocol, the authentication type, and the encryption method employed between two hosts, in general the following services are provided at the session management layer.

Before allowing a network liaison to commence, the TCB's of the communicating systems perform a network handshake authenticating themselves to each other, and the liaison participants to each other. By selecting the type of session management (SM) to be used with a particular host, the NSO controls:

- Whether or not there is an actual network handshake
- Which protocol is used for the handshake
- Whether authentication is optional or mandatory
- Whether authentication may be requested by the application layer

By selecting the type of authentication, the NSO determines which authentication mechanism should be used by the handshake for verifying the two sides.

During the authentication handshake, the TCB on each side of the network liaison determines whether or not to let the liaison continue based upon the security attributes of the liaison participants, the requested level of the liaison, the accreditations of the two hosts involved, and the security policies enforced on the local system. After approving a network liaison, the security attributes associated with each packet imported or exported for that liaison are checked to ensure that they are within the approved set for that liaison.

To send data to a remote host, the sensitivity label of the data must be within the accreditation range specified for that host (as defined in the Remote Host database). In order to receive data, the sensitivity level of the incoming data must be dominated by the receiving process and it must be within the accreditation range specified for the remote host. Some types of session management may enforce additional checks, and are thus dependent upon the NSO's choice of SM protocol to use with a particular remote host.

Attribute transport is an essential service that is required in order to support the access control and authentication services. Unless this service is fully supported between two hosts, communications between them are not trusted and should only be permitted after special consideration by the NSO. Attributes may be transported at the network layer as well as by the session management protocol for those routing attributes. If all attributes are transported at the network layer, then none need be transported at the session management layer. Any attributes transported at both layers will be chosen from the session layer set. Whether or not attributes are transported at the session layer depends on the type of SM chosen by the NSO for communications with a remote host. This service is always available at the application layer in order to support trusted network applications. The set of attributes actually transported between applications depends on the security configurations of the two machines involved.

Network auditing in MAXSIX is independent of the underlying session management type, although some SM protocols may generate audit records unique to that protocol. The NSO selects which set of network events to audit, and whether the events should be audited locally or sent to a network audit collection center.

The Network Level Module provides a number of services that include the following.

- Attribute Transport: The NSO specifies the set of attributes to be transported and the labeling format to use at the network layer for communications with each remote host. This choice should be compatible with the type of SM chosen for that host, for some SM protocols require specific network labeling formats.
- Access Control: This service requires the network layer attribute transport service to properly label each incoming/outgoing packet with the mandatory access policies configured between

the sending and receiving systems. In order to send a packet, the attributes of the packet must lie within the accreditation of the interface upon which the packet will be transmitted. In order to receive a packet, the attributes of the packet must be dominated by those of the receiving interface. This service is not configurable and is always enforced. However, the service's ability to enforce these requirements on incoming packets is dependent on the protocols chosen by the NSO for communications with the remote host.

- Trusted Routing: This extension to the access control service ensures that if a packet cannot be delivered directly to its destination, it will only be routed through gateways and networks accredited to receive the packet.

Most MAXSIX functionality is contained inside the kernel and is hidden from the Network Security Officer, as well as from the user. What the NSO sees are the databases, the MAXSIX daemon, and the utilities used to manipulate it. Once the databases and other components are installed and working properly, the NSO normally will not need to deal with the trusted portion of the network.

4.2.2 Trusted Mach [References: 28, 29,30, 31, 32, 33]

4.2.2.1 Overview

The Trusted Mach (TMach) system is a server-based operating system implemented as a message passing kernel and a set of servers that was developed by Trusted Information Systems (TIS) located in Glenwood, MD. (TIS was also responsible for the Trusted Xenix implementation.) An extensive documentation set has been developed for TMach in the form of executive summaries and development of the system continues to be supported by DARPA. The TMach system servers provide operating system functionality that is traditionally found in the operating system kernel. The servers are classified as trusted or untrusted. The kernel and the trusted servers comprise the TMach TCB, which provides a personality neutral set of services. Personality neutral refers to limited subset of generic capabilities provided by the TMach kernel. These functions limited in scope, independent of any particular UNIX operating system implementation, and are meant to provide basic services for constructing other higher level services. The untrusted servers use these services to present a more complete operating system personality to the users of the system.

The TMach System is designed and implemented with several primary goals, which include:

- to be evaluated at the TCSEC B3 level and the ITSEC F-B3 functionality class, high minimum strength, and E5 evaluation level.
- to be easily portable to a wide variety of hardware platforms, so long as those platforms provide a minimum set of facilities.

- to perform competitively with other commercially available systems, e.g., TMach with UnixTM emulation compared with an untrusted BSD Unix implementation.
- to be easily extensible, for example, by allowing the addition of a different user-level interface, a different kind of file system, or new device drivers.
- to support a POSIX interface, ideally one binary compatible with an existing untrusted POSIX system.

The basic architecture of the TMach system consists of:

- The TMach kernel which provides the basic functionality of the TMach system.
- TCB servers which provide a set of personality-neutral, multi-level secure services. The clients of these services include TCB servers themselves and untrusted software components. The TCB servers use the basic features of the kernel to implement higher level functionality and enforce the TMach system security policy controlling access to those functions.
- The untrusted code in the TMach system provides the user-level interface, also called the operating system (OS) functionality. In providing this functionality, the untrusted code may use the TCB services as little or as much as required. However, in no case can the untrusted software bypass or subvert the TMach sÿstem security policy.

In addition to using the kernel to build higher level functions, the TMach TCB servers also use the functions to protect themselves from each other and from untrusted code. Each TCB server is a separate task, with its own separate virtual address space and port space. The only way two TCB tasks can communicate with each other is via the kernel IPC and shared memory features. This separation provides very strong modularity within the TMach TCB. Each server is a self-contained execution environment and can only be affected by other servers through a well-defined interface.

Each TCB server also has a unique user id. This user id is part of the server's security identity and is used to mediate the server's access to items in the TMach system name space. The TCB closely controls the creation of tasks with TCB identities to guarantee that these identities cannot be spoofed.

Access mediation in the TMach system is centralized in the Root Name Server (RNS). All named entities are controlled in the TMach name space. The RNS manages the name space and holds all security-relevant information about the named entities. The RNS makes all mediation decisions based on the TMach system security policy. There are several types of named items in the TMach system, e.g. directories, files, virtual terminals. Each type has its own item manager server which maintains the contents of the items of that type and implements type-specific operations.

The item managers enforce the mediation decisions made by the RNS. Once the RNS approves an access request, it forwards the request on to the appropriate item manager. Along with the information identifying the specific item, the RNS gives the item manager the identity of the requesting client and the types of access that were granted for the item. If the request is from a different client or requires access not already granted, the item manager refuses the request.

TMach contains a centralized Class Library that provides a common framework for associating clients with access requests and for performing access control enforcement. The library routines associate the client's identity and access with the active connection. As requests come in over the connection, the Class Library routines perform enforcement checks and only pass on permissible requests to the specific item manager routines. The Class Library routines also perform IPC management.

In TMach, the Subject Server is the only TCB server that can directly create new subjects. All other TCB servers that create new subjects must use the Subject Server. The Subject Server is the central point of control for creating new subjects. The Subject Server implements a policy which does not allow any task to create new subjects with TCB identities. There is a small set of specific cases that violate this policy.

4.2.2.2 TMach System Architecture

The TCSEC and ITSEC requirements mandate use of modularity and the application of Least Privilege. To meet both of these principles requires the separation of functions. However, too much separation can often introduce performance penalties. The TMach system architecture attempts to reach a balance between the separation and structure required by the security criteria and performance.

The TMach system security policy is centralized in the Root Name Server (RNS). All access to named items begins with requests to the RNS. All other servers, both within and outside of the TCB are subject to the security policy. Since the RNS implements the TMach name space, there is no way to bypass the RNS access checks and gain access to named items. By centralizing the security policy in a single server, analysis of the policy is simplified.

Initial access to a named item involves both the RNS and the item manager. However, requiring both servers to be involved in all requests would place an unacceptable performance penalty on the system. To counter this problem, the item managers have the responsibility for enforcing the

RNS access decisions. Upon initial access, the RNS passes information sufficient for enforcement to the item manager. All subsequent requests to the same item can be sent directly to the item manager, which can enforce the original access decision, using the information provided by the RNS.

The TMach RNS itself is a special server, in that it not only implements the system name space and system security policy, but it also implements item management for several types of items - directories, mount points and symbolic links. The RNS also implements the Security Id service, which translates each task's security identifier into a specific set of security attributes, i.e., security labels, a user id and a list of group ids. The Security Id service was implemented as part of the RNS for performance reasons.

The TMach kernel exports three IPC ports - the host security port, the host control port and the master device port, each of which has a send right associated with it. Possession of a send right to one of these ports grants the holder of the right the ability to make all legal requests identified with the port. In the TMach system, several servers need to make requests over these ports, but no server needs to make all the requests. Rather than give rights to these ports to each server that needs to make any call on them, the TMach system includes three special TCB servers, one for each port. These servers each hold exclusive access to their respective port and can decide whether to perform requests to the port on behalf of other tasks. The Subject Server holds the host security port and implements a policy controlling specification of security ids for new tasks. The Hôst Control Server holds the host control port and decides which TCB tasks can perform which host control commands. The Device Server holds the master device port and issues kernel device ports based on the device configuration in the system name space.

The Trusted Shell (TSH) and Trusted Administration Shell (TASH) Utilities are groups of programs which implement the TMach trusted path and trusted administration functions.

The separation of the TMach TCB into a number of servers with unique identities provides the ability to expand the TCB in many different ways. Centralization of system security policy in the RNS makes it possible to change the security policy without affecting the item managers. Similarly, the addition of a new item manager is simply a matter of writing a new server for the system. The existing TCB servers need not be affected by the new item manager at all. Additional TSH or TASH Utilities may be added in the same fashion. The use of separate tasks for TCB servers, as well as careful design of the interfaces between servers, allows the addition of new TCB tasks without compromising the security of the existing TCB servers.

By providing personality neutral TCB services, multiple OS personalities can coexist on the same system. The TMach TCB ensures that the untrusted code that imparts personality to the system from the users perspective does not violate the system security policy. From within those constraints, however, the OS personality can provide need user interface functionality.

The TMach kernel is the lowest and most primitive layer of the TMach architecture. It provides the basic mechanisms upon which the more familiar operating system abstractions are built. For this reason, the kernel provides a few simple, but powerful, abstractions that enable the higher architectural layers to construct more numerous and complex abstractions.

The TMach kernel provides the following abstractions, as expected from an operating system derived from Mach:

- Task and Threads: Like Mach, threads are the executing entities on the system. Tasks, which are composed of one or more threads, provide the environment required to execute thread(s). The primary resources provided by a task are ports for intertask communication and memory.
- Ports and Messages: Messages are the primary means by which tasks communicate. Messages are passed between tasks through ports. Ports have the ability to queue messages. Individual tasks can obtain port rights, which enable the task to enqueue or dequeue messages from ports.
- Memory: Memory objects are storage areas from which data can be read or written. Tasks obtain rights to read and write data from memory objects.

In addition to these primary abstractions, there are additional abstractions provided by the kernel. The most important of these additional abstractions are I/O devices. Although I/O devices are not critical to the functioning of the kernel, they are very important to the TMach system as a whole because the operating system is not particularly useful without them.

The kernel does not directly implement the TMach security policy. Instead, the TCB servers use the hardware features, kernel features and services to construct the security features and functions necessary to implement the TMach security policy. One particularly important feature provided by the kernel is the security id associated with each task. The TMach TCB servers use this identifier to identity the user executing the task.

The TMach system is designed to be easily portable between different hardware platforms. The TMach kernel is divided into hardware-dependent and hardware-independent portions, allowing replacement of the hardware-dependent code without requiring changes to the hardware-independent code. These two portions of the kernel communicate through a well defined interface. Port-ing of TMach to other platforms is simplified by this architecture.

The TMach TCB is layered on top of the TMach kernel. The components of this layer are trusted servers that use the basic features of the kernel to provide a personality neutral set of services to untrusted software.

The servers in the TMach system perform many common functions and they make use the kernel IPC facility to implement remote procedure calls (RPCs). Most of the servers are item managers managing items in the name space. The servers also perform a set of common functions related to managing threads, translating messages, handling RNS requests, and executing routines to service the requests. The Class Library is a collection of routines that implement these common functions so the TMach servers can perform these functions in a standard way.

The TMach Class Library provides the following services to the TMach servers:

- RPC: The RPC routines provide a framework that hides the fact that the procedure call is remote. Services are provided by the TMACH Class Library that translate the arguments into messages and back into arguments as necessary on both ends of the RPC call.
- Port Management: Services are provided by the TMACH Class Library that manage the port space of a server, creating and deleting ports as necessary, setting port attributes and matching ports to items during RPC handling.
- Thread Management: Services are provided by the TMACH Class Library to control the number of threads used service incoming RPCs by automatically creating and destroying threads. Basic locking mechanisms are also available to allow for thread synchronization.
- Item Manager Support: Many of the [illegible] servers are item managers. Services are provided by the TMACH Class Library for it[illegible]anagers to register with the RNS, to communicate with the RNS about item creation, to delete, to access, and to enforce RNS access control decisions on incoming requests. The enforcement includes verifying that the correct client is making the request and that the RNS has granted the client the type of access required by the request.

Although it is not a separate task, the Class Library is the basis for all of the TMach TCB servers.

Several TMach TCB servers access disk storage devices. For example, the Root Name Server and File Server manage data on disks, the bootstrap programs read files from disk before the Name and File Servers are active and certain utility programs access disks drives directly to verify the integrity of TMach Name and File Server data. The TMach Disk Library provides a common interface for low level access to the disk devices via calls to the TMach Kernel's device I/O interface. This interface hides the physical structure of data on disk and provide a high-level interface for accessing and managing disk space. The disk library exports two main services:

- File: A file is a logically contiguous stream of bytes. The file services support operations for reading data, writing data and truncating a file. The file operations hide the explicit details of allocating, freeing and remembering the location of disk blocks.
- Partition: A partition is a pool of disk blocks available for file allocation. The partition service supports operations for creating, finding, closing and destroying files.

The Root Name Server (RNS) resolves names and mediates access for the system name space. In addition, the RNS implements several types of items associated with the name space. Finally, the RNS implements the Security ID service. Every named item in the TMach system is part of the system name space. The individual named entries are called items. Each item is an instantiation of a type, e.g. directory or file, and each type is itself an item in the name space.

An item's entry in the name space contains all the information necessary to mediate access to the item, such as the item's security label, its access control list (ACL) and its type. The RNS also stores other information which is common to all types of items. This common information includes creation and modification dates, a link count and information about the item's location in the name space. The RNS stores its database on one or more disk partitions designated solely for this use. The RNS implements a Bell- LaPadula security model and uses information about an item's type to translate requested access operations. DAC is performed by comparing the client's user and group ids and access request against the item's ACL.

While the RNS mediates access to all items, the contents of an item may only be retrieved by communicating with the item's manager. The RNS itself is the item manager for several types closely associated with name space management: directories, mount points, symbolic links and types. All other types, e.g. files or terminals, are managed by separate servers, called item managers. The RNS and an item's manager work together to provide all information about the item. In other words, the RNS provides service for an item's name while the item's manager provides service for the item's contents.

The TMach kernel exports a host security port that allows creation of a new task with an arbitrary security id. Without this port, a task may only create new tasks which have the same security id as the parent task. In the TMach system, the security id is used as a subject id.

The TMach kernel exports a master device port which allows access to all devices on the system.

The Device Server is the only task in the TMach system which holds send rights to this port. In response to requests mediated by the RNS, the Device Server passes send rights to the device ports to the servers which manage the devices.

The File Server (FS) provides access to the contents of files contained in the file system. The FS knows about the structure of the file system as it is stored on disk. The File Server is implemented as an external pager. File I/O is a series of memory references, with the FS paging the file into physical memory as needed. The resulting client interface is simple.

The Authentication Server manages a set of databases used by the TMach system for identification and authentication. The databases also contain the translations between machine readable and human readable forms of various data, such as user and group ids and security labels.

The Audit Server is the central collection point for all audit data in the TMach system. The Audit Server registers with the RNS as the item manager for the audit type. TCB servers open instances of the audit type in order to submit audit data. As auditable events occur, the TCB servers collect the necessary information and send audit requests to the Audit Server, which in turn writes the audit events to the audit file.

The Virtual Terminal (VT) Server is an item manager which manages the physical terminal devices on the system. Clients can then create single level virtual terminals by creating instances of these types. The actual physical terminal associated with a virtual terminal is determined by the type used to create the virtual terminal. The VT Server also scans the input stream for the Secure Attention Key sequence (SAK). The VT Server sanitizes and resets the physical terminal each time it activates a different virtual terminal.

The Trusted Shell (TSH) Utilities consist of the Trusted Shell program and a set of utility programs which implement trusted path commands. Unlike the TCB servers described up to this point, the TSH Utilities are not a single instance of a server providing service to the entire system. Instead, a separate TSH program is instantiated for each terminal on the system, and TSH utility programs are executed on demand from the trusted path.

TMach administration encompasses a large set of utilities necessary to maintain, configure and manage the system. These utilities are available via the trusted path to provide assurance that the system administrators that they are communicating with the TCB. The utilities are divided into several roles, the Security Operator or the Auditor. Each of these roles has a specific set of commands associated with it. The trusted administration roles supported by the TMach system are:

- Trusted System Programmer installs the system and performs non-routine maintenance

Constraints and System Primitives in Distributed Systems

- Account Administrator configures and maintains the accounting databases
- System Security Administrator configures devices and file systems, maintains authentication databases, performs routine maintenance
- Auditor configures and manages system auditing, analyzes audit data
- Security Operator starts and stops system, performs backups, enables/disables printers, etc.

5.0 Lessons Learned from MLS/GDSS Project

This portion of the report focuses on several important issues that were identified in the MLS/GDSS project and are relevant to the development of distributed trusted operating systems.

5.1 Introduction

In the Spring of 1989, a Multi-Level Security (MLS) testbed was initiated by the United States Transportation Command (USTRANSCOM) in conjunction with the Military Airlift Command, now the Air Mobility Command (AMC). The testbed's primary mission was to demonstrate the feasibility of developing MLS military systems by re-engineering an existing Command and Control (C^2) system, the Global Decision Support System (GDSS) into a multi-level secure GDSS (MLS/GDSS). The first phase of operational deployment of MLS/GDSS occurred in September of 1993.

The GDSS is the primary command and control system of the AMC and provides flight following, mission planning and scheduling, logistics, transportation, and personnel support capabilities. In addition, the GDSS communicates to sixteen other external systems and is distributed at several sites around the world. The MLS/GDSS system, which will replace the existing system, is targeted to provide a B2 level of security assurance. It will be capable of handling sensitive unclassified through secret data and will provide all of the functional capabilities of the currently deployed system. Because it is a C^2 system, the MLS/GDSS not only must perform the fundamental information management operations found in most information processing systems, it must also ensure a very high level of integrity and be capable of handling widely varying levels of transaction processing in a timely manner.

5.2 MLS/GDSS System Design Goals

Some of the overall software architecture goals of the MLS/GDSS system included:

- Build a system that performs the same operations as the currently deployed system in a secure fashion and is capable of processing, storing and protecting Sensitive Unclassified through Secret information when the user community consists of both cleared and uncleared individuals.
- Use Commercial-Off-The-Shelf (COTS) and Government-Off-The-Shelf (GOTS) products that conform to accepted standards.
- Design and build a system that provides trusted labeling of single data elements on the user screen.

This section of the report contains a discussion of our efforts to design a system architecture that meets the above requirements, and focuses on attempts to provide a means of protecting data and associated labels throughout the system. The discussion introduces the concept of a Trusted Data Distribution Pathway (TDDP) that is composed of the Trusted Computing Bases (TCBs) of COTS products and additional trusted code that was developed to integrate these components. This pathway delivers trusted data element-level sensitivity labels to the user interface. The system COTS components include a trusted MLS operating system, trusted MLS relational database management system (RDBMS), trusted MLS network, and trusted MLS user interface.

5.2.1 Trusted vs. Advisory Labeling

Currently, there is debate over the need for the display of trusted sensitivity labels to the user. In the MLS/GDSS system, it is possible for both disclosure and command decisions to be made based on the classification of single data elements. A significant amount of standard command and control activity involves computer-based applications executed in conjunction with secure and unsecure phone and HF radio communications. Users of the currently deployed GDSS system identified this need very early in the process of developing system requirements for the replacement MLS/GDSS. Our analysis of several other command and control systems indicates that the need for data element level labeling is a common long-term requirement of C^2 systems.

As systems integrators developing the MLS/GDSS system, we have had difficulties in understanding why displaying advisory labels would be advantageous if trusted labels could be made available to the user. If the label of a data element cannot be trusted to be accurate, the display of any label other than that of the session level is confusing to the point of danger. It is clear, as discussions with system users indicate, that if any sort of decision, command or otherwise, is to be made based on the label of a data element displayed to the user then the label must be trusted. Otherwise the user must treat all data as if it is labeled at his session level. The display of advisory labels to the user could have serious consequences, particularly if the user is inadequately trained in the use of advisory labels.

From our perspective, in rapidly reacting command and control systems like the GDSS, display of untrustable data, i.e. advisory labels, to the user worsens his work load and may complicate his decision making. We realize that this is an extreme position and may not be applicable in general to all systems. In particular, advisory labels could play an important role in information analysis systems. Our experience with C2 systems, however, points out the absolute requirement for trusted labeling of data elements in command and control systems.

5.3 Trusted Data Distribution Pathway (TDDP)

A major obstacle in fulfilling the goals of the program was the design and implementation of a complete protected pathway for the movement of data and labels between the RDBMS, network, and the user interface. This component of the system is known as the Trusted Data Distribution Pathway (TDDP). The TDDP encompasses all of the TCBs of the COTS components that comprise the system and all trusted code that was developed to compose these products into a functional system. Thus, the TDDP is a union of all the TCB components that form the system.

5.4 TCB Composibility

In designing the system using COTS products, a number of shortcomings were identified in the ability of these products to be composed to form a TDDP. These shortcomings were overcome by the development of additional trusted code designed to extend the TCBs of the COTS products and to complete the TDDP.

To provide a TDDP that permits the user to trust data sensitivity labels displayed on the screen, all of the individual TCBs of the COTS products must be connected to each other in a secure manner. Each product must also provide a complete and trusted pathway to manage data and associated labels within its own TCB.

As an example, consider an MLS RDBMS product. Most of the products that are currently available include a trusted relational database engine that forms the core of the RDBMS, and associated Application Programming Interfaces (APIs). Many of these products also support a client-server model, with the RDBMS engine capable of residing and one machine and a API component resident on a different machine. Presumably the two machines are connected with a trusted MLS network.

Much of the focus of the vendor's development, and indeed the evaluation process of these products, is on the relational engine, its security policy and TCB. In order for these products to be used to develop systems that provide trusted data labeling, each component of the product having access to the data and labels must be evaluated for trust. Not only must the relational engine be secure, but its interface on the server and the API components on the client machine must also be trusted. Thus, the TCB of the RDBMS must encompass all of these components. Unless all elements are secure, the MLS RDBMS can at best only ever provide advisory labels, forcing the classification of all data displayed to users in an MLS environment to be treated as session level. Systems which provide trusted labeling unavailable if only the relational engine forms the TCB of the RDBMS products. Unless there is a trusted means of exchanging data and associated labels between the trusted application code and the API of the RDBMS, the labels are only advisory.

Requiring all elements to be trusted imposes difficult constraints on the product vendors and on the evaluation process. RDBMS products are currently treated as layered system components and evaluated against the Trusted Database Interpretation (TDI) using a particular combination of RDBMS and underlying operating system. In a client server architecture, or even when all components of the RDBMS are resident on the same system, the MLS network should also must be considered in the evaluation process if these products are ever intended to be used to develop systems providing trusted data labeling.

5.5 Protected Data and Labels Storage

One of the obvious requirements that must be met by the TDDP is the protection of data and associated labels as they traverse the TDDP. At no point must untrusted components of the system be permitted access to the labels in such a way that the labels might be modified in a manner that violates the security policy enforced by the system.

We assume that the TCB of the evaluated COTS products provide adequate protection of labels. A major shortcoming in using available COTS products to build systems similar to the MLS/GDSS is in the area of the user interfaces, as discussed in more detail later. In composing the system from various TCBs, it was necessary to develop a trusted repository for data and labels available to the applications that make up the system. Once the data and associated labels leave the RDBMS and are delivered to the application requesting the data, they must be protected. This trusted repository provides secure memory management capabilities to the applications.

Once the data and labels enter the address space of the application, they are accessible to any code executing within the user's process. Without a protected memory manager, all application code would need to be trusted not to violate security policy and accidently or deliberately modify labels. The executable files that make up the MLS/GDSS applications are built from sources including more than 1 million lines of Ada code. Requiring trust of this amount of code is not feasible. Use of a segregated and protected memory manager helps permit the development of applications in a secure MLS environment without requiring the applications to be trusted. Thus, the secure memory manager provides intra-process protection of labeled data.

5.5.1 Security Policy Enforcement

Another significant issue in composing COTS products into a secure systems capable of displaying trusted labels is that of consistent security policy enforcement. When incorporating multiple TCBs into an integrated TDDP, it is obviously important to ensure that all of the components enforce the same security policy. Fortunately, the security policies of currently available products all support a modified Bell-LaPadula model. Most of the products do not permit write-up of data, although this is allowed under the original model.

In composing these COTS components into an integrated system, it was necessary to support a similar model within the trusted code components that were developed to support applications development. The overall security policy developed for the MLS/GDSS systems also incorporates a number of requirements, such as data ownership rules, downgrading rules, and auditing rules that are not inherent in the TCB components. The additional trusted code developed to support a TDDP is responsible for enforcing these components of the security policy within the process space of the user. Through the programming interfaces to the COTS products, trusted code developed to compose the system also ensure that these requirements are supported by the TCB of the appropriate COTS product.

One example concerns the generation of data created in the user's process. Any data that is extracted from the RDBMS is labeled by the RDBMS when first entered into the relational engine. However, data directly entered into the user interface forms or by the application code processing this new data must be properly labeled. The MLS/GDSS security policy stipulates that any data created within the user's process must be labeled at the level of the user's process. This rule is enforced by the trusted memory management component when memory is allocated for the storage of new data.

It is possible, for example, for a user to enter new data into a screen form and then request the system to display the sensitivity of the data before the data is actually stored in the RDBMS and properly labeled by the TCB of the RDBMS. The same is true of data created to build the appropriate tuples to insert into the RDBMS. New data entered into the system by the user through the user interface may cause the application to generate several tuples of data to support the relational model.

Addition of trusted code to complete the TDDP was necessary because the TCBs of the COTS products used do not extend to the application's domain. The TCB of the RDBMS ends when data is transferred out of the API of the RDBMS. The TCB of the operating system enforces labeling of data entering or leaving the user process and enforces inter-process communication labeling. There is no support for intra-process security policy enforcement or protection by the TCBs of any of the COTS products. For this reason, it was necessary to develop additional trusted code.

5.5.2 Data Classification Constraint Enforcement

Another requirement that was identified in designing the MLS/GDSS is the need to support data classification constraints. Classification constraints are rules that stipulate the levels of classification are permitted for data contained within specified fields. While most of the COTS database products permit the enforcement of such constraints within the RDBMS, no generalized support exists outside the RDBMS TCB. In composing the system, it was necessary to add trusted code that supported these rules throughout the TDDP.

For example, if the user operating at a SECRET session attempts to enter data into a field constrained to hold only UNCLASSIFIED data, the resulting classification constraints must be immediately enforced. Operationally, it would not be feasible to allow the user process at the user interface and application level to violate security classification constraints, and then enforce these rules inside the TCB of the RDBMS. Violation of the classification constraints rules during data entry into the user interface could inadvertently lead to inappropriate command decisions. Only when the user attempts to commit the data to the RDBMS would the constraint violations be detected and reported. Unless the constraint violations are enforced when that data is entered into the user interface, the user could incorrectly make a decision based on the classification of this data since the rules are not enforced until the user commits the data to the database. It is, therefore, necessary to ensure that the TDDP enforces the same security policy throughout the pathway, and not just within the TCB of a single component.

5.5.3 Data Element Level Labeling

As stated before, one of the requirements of the MLS/GDSS is to provide labeling at the granularity of single data elements. This granularity of labeling, however, is not supported by the currently emerging MLS RDBMS products. The RDBMS products provide tuple level labeling which is translated to single data element level labeling by trusted code that interfaces to the RDBMS API and uses the protected memory manager of the system. The code uses a process called row-collapsing and polyinstantiation to support data element level labeling and cover stories. The design of these components has been described elsewhere.

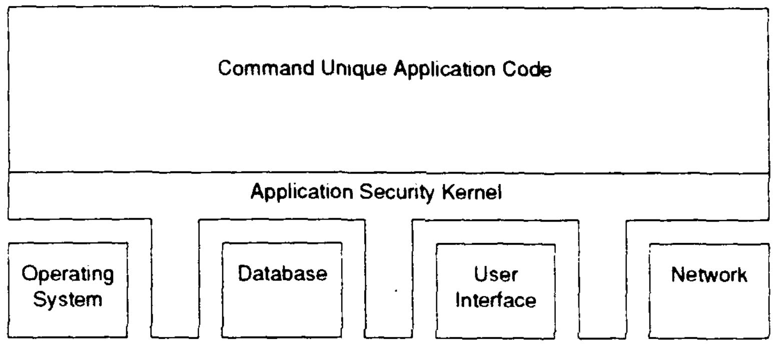
Since this component has the service to modify labels directly, it must be trusted. As data and associated labels are received from the RDBMS API labeling granularity is translated from a tuple-level to data element level, and the resultant data set transferred to the protected memory manager. The interface is constructed from trusted code that was developed to compose the system using COTS RDBMS products. Its translation of tuple-level labeling to data element level labeling assumes that the underlying RDBMS securely transfers data and associated labels from the data structures within the RDBMS's API, and those data structures allocated by the database interface code to receive them.

5.5.4 The Application Security Kernel (ASK)

In order to provide a TDDP, and to provide isolation of the TDDP from the command unique applications, we developed an architecture that provides a layer of TCB extensions residing between the applications and underlying COTS products.

The core of the system is called the Application Security Kernel (ASK) which fulfills two very important, basic functions. First, the ASK provides an isolation buffer between the Command Unique Application Code and the Trusted Products which form the majority of the Trusted Computing Base (TCB). Second, it provides extensions to the TCB that enforce the security policy of the system, as well as extensions to the capabilities of the COTS products to meet the functional requirements of the system. The relationship of the ASK to the other major components of the system is illustrated in Figure 1.

Figure 1:The MLS/GDSS Software Architecture



The ASK consists of a body of procedures, written in C, which serve as an interface between the Applications and the Trusted COTS Products. Its primary purpose is to provide a complete set of application programming services that unconditionally implement the security policy. No sequence of operations which the applications can perform will cause the security policy of the system to be violated. As such, the application code does not have to be treated as trusted code during the security evaluation process. This is key to achieving accreditable security in the system, since the security kernel represents just 1% of the total code in the system. Thus, the focus of the accreditation process may be *significantly* narrowed.

The ASK provides several critical capabilities to the system. Included in these capabilities are the following:

- A complete TDDP that includes a protected memory management component that is available to the applications and other ASK components. This repository protects the data and associated labels within the context of the user's process. In conjunction with the TCBs of the

COTS products the TDDP provides a trusted pathway for transfer of trusted labels to the user interface.

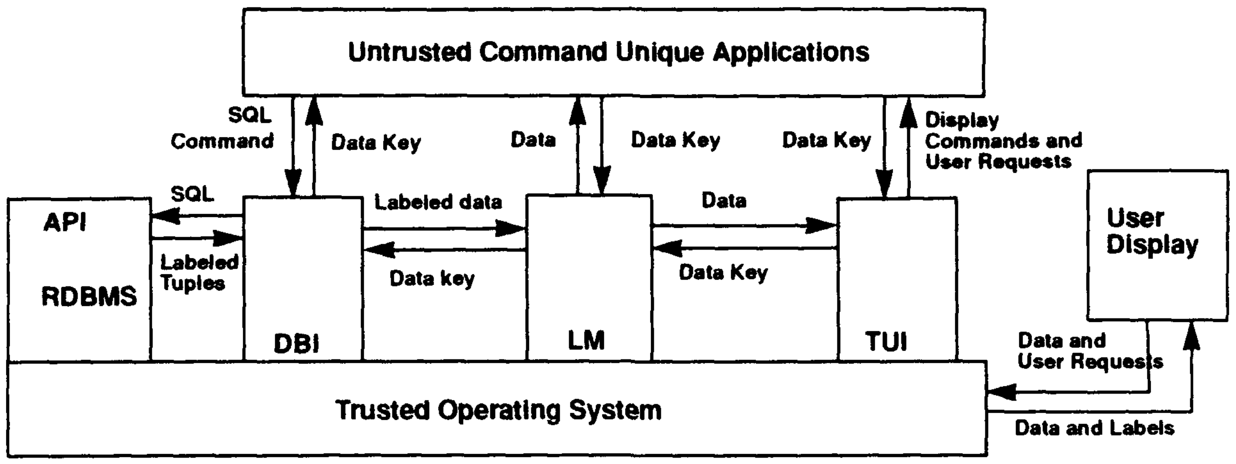
- Trusted computing base extensions enforcing the security policy of the system and providing security functions to the applications through an open library containing numerous functions. These functions include support such as label translation between the various COTS components that comprise the system, trusted memory management of data structures used by the untrusted applications, and trusted access to various services provided by the COTS products.

- Extension of the capabilities of the COTS products to meet both security requirements and functional requirements of the system. The security extensions include: auditing capabilities of the system's COTS products to meet the requirements associated with a B1/B2 trusted computing systems, and additional requirements needed for command and control systems such as GDSS. An example of a functional extension is the ability to provide data element level labeling to the user. Current MLS database management systems provide, at best, tuple level labeling capabilities. Because of the need to make disclosure decisions regarding the values of individual data elements associated with a mission (such as landing time) to unclassified users of the system, it was necessary to extend the labeling capabilities of the system to the granularity of single data elements.

5.5.5 Processing of Application Commands by the ASK

Figure 2 shows an expanded view of the COTS components and interfaces provided by the ASK. The shaded sections of the diagram indicate those components forming the TDDP. Three of the major components of the ASK are illustrated: label manager (LM), database interface (DBI), and trusted user interface (TUI). The command unique application code comprises the majority of the system's developed code, and is not trusted to enforce ant component of the security policy.

Figure 2:Process Flow in the TDDP and ASK



When the user, through keyboard commands, requests a display of database data, this request is passed to the application code, which issues multiple query strings to the database interface component of the ASK. The ASK composes these strings into SQL commands and issues these commands to the RDBMS. The ASK then processes the returning rows to provide data element level labeling and passes the resulting data set to the label manager. The label manager allocates protected memory to hold the data and labels, and returns a data key to the database interface. This key is in turn passed back to the application.

The application is free to request copies of the data, make decisions based on the data, or generate new data requests in preparation for displaying data to the user. The application then instructs the trusted user interface component of the ASK to display the data by passing the appropriate data keys. The user interface assumes control, requesting the data and associated labels for display directly from the label manager using the data keys. The user interface then processes commands from the user. When the user modifies existing data or enters new data and requests processing of the form, the user interface transfers the data to the label manager, that enforces the proper labeling of the new or modified data elements and returns a data key to the user interface. This new set of data keys is then transferred back to the application. At no point does the untrusted application code have the ability to modify the labels associated with the data. With the exception of the RDBMS server, all of the components operate within the context of a single process.

5.5.6 Trusted User Interface COTS Products

The MLS GDSS system is primarily a forms-based application; users are presented with forms containing information read from the relational database. Users have the ability to modify the information and store the modifications in the database. The user interface in a MLS system has to enforce the security policy of the system. In the case of MLS GDSS, the user interface has to:

- Show the user the security label of every piece of information on the screen
- Ensure that data entered by a classified user is marked as classified even before the data is entered into the database
- Provide the capability of covering up all of the classified data on the screen
- Provide a trusted path between the user and the remainder of the system
- Support character cell and X-based terminals
- Provide a migration path to windowed workstations.

At the present time, there are no commercially available user interface products that meet these minimum requirements. However, the FIMS standard is available as a forms management standard, to which any future trusted forms product is likely to adhere. At the present time, there is an emerging B2 compliant product being evaluated by the MLS/GDSS project.

5.5.7 Summary

This section of the report has presented some of the issues regarding provision of trusted labels in the MLS/GDSS, a system which is being built using Commercial-Off-The-Shelf (COTS) products and accepted government and commercial standards, to produce a portable system design and architecture that provides B1 and B2 levels of security for command and control systems. As a result of the effort, the highly portable system architecture of the Application Security Kernel (ASK) provides security extensions and seamlessly integrates a number of secure COTS products into a functional information management system. The ASK is designed to support a Trusted Data Distribution Pathway which permits protected management of data and associated labels, and ensures trust of sensitivity labels displayed to the user. The AMC is currently using this architecture and software to develop several secure C^2 systems.

The design and development of the ASK was driven directly by shortcomings in the existing MLS COTS products, particularly the existing MLS operating systems. The lack of functionality and support for developing true MLS applications in existing MLS operating systems severely limits the potential development and accreditation of MLS systems and MLS software products. If MLS operating systems evolve to provide:

- support for developing trusted applications

- true MLS data and file structures
- greater labeling granularity and
- a trusted data distribution pathway

we fully expect the need for the ASK to disappear. If there an evaluated MLS operating systems existed which provided these services and met a B2 or better level of assurance then the ASK would be unnecessary for development of MLS applications.

6.0 Security Threats in Distributed Operating System Environments

In addressing overall security in a distributed system, it is necessary to integrate computer system security and communication security measures to protect information both within the system and between systems. Neither one on its own can provide the required complete protection of information in a distributed environment. For instance, access controls to restrict users gaining access to a resource within a system or on a network together with suitable flow constraints to regulate the flow of information are essential. Trusted computer system mechanisms are needed to ensure the enforcement of security controls and in the provision of the necessary assurance that the correct operation of the security measures are maintained. Secure protocols are vital to the successful operation of security measures. Security mechanisms like encryption algorithms form an essential part of the overall solution.

6.0.0.0.0.1Security Threats

This section describes a list of security threats which can arise in a general distributed environment. This discussion takes into account the threat statements from various standards documents: ISO 7498-2, X-400, ECMA TR/46, and DAF Security.

- Masquerading: One entity pretends to be another entity. By masquerading, an entity can obtain privileges which it is not authorized to have. This can happen at different levels in a distributed system: within a computer system, a user or process might masquerade as another, to gain access to a file or memory to which it is not authorized. Over a network, a masquerading user or host may deceive the receiver about its real identity.

- Unauthorized Use of Resources: This includes unauthorized access to resources on the networks, as well as within a system. For instance, within a computer system, this threat corresponds to users or processes accessing files, memory, or processors without authorization. Over a network, one threat is accessing a simple network resource, such as a printer or a terminal, or a more complex one such as a database, or applications within the database. Unauthorized use of resources may lead to theft of computing and communications resources, or to the unauthorized destruction, modification, or disclosure of business information.

- Unauthorized Disclosure and Flow of Information: This threat involves unauthorized disclosure and illegal access of information stored, processed, or transferred in the distributed system, both internal and external to the user. Within the system, such an attack may be unauthorized reading of stored information, while over the network, the means of attack may be wiretapping or network traffic analysis.

- Unauthorized Alteration of Resources and Information: Unauthorized alteration of information may occur both within a system (by writing into memory), and over the network (through active wire-tapping). The latter attack may be used in combination with other attacks such as replay whereby a message or part of a message is repeated intentionally to produce an unauthorized effect. This *threat may also involve unauthorized introduction or removal of* resources into or from a distributed system.

- Repudiation of Actions: This is a threat against accountability in organizations. For instance, a repudiation attack can occur whereby the sender (or the receiver) of a message denies having sent (or received) the information. For instance, a customer engages in a transaction with a bank to debit a certain amount from his account, but later denies having sent the message. A similar attack can occur at the receiving end. For instance, a firm denying the receipt of a particular bid offer for the tender even though it actually did receive that offer.

- Unauthorized Denial of Service: Here, the attacker acts to deny resources or services to entities authorized to use them. For instance, within a computer system an entity may lock a file thereby denying access to other authorized entities. In the case of the network, the attack may involve blocking the access to the network by continuous deletion or generation of messages so that the target is either depleted or saturated with meaningless messages.

6.1 Security Mechanisms

For a distributed environment, one needs to determine which threats are applicable. The overall set of security measures required to counteract the identified threats constitutes the security policy. A security model should be applicable for a wide range of systems and applications, and is intended to include a wide range of security services that can be used and combined in different ways to meet different security policies. In particular, a distributed environment is likely to have multiple security policies, and different components responsible for the various parts of the system. A sample set of such services is listed below:

- Identification and Authentication: This service provides the confidence that at the time of request, an entity is not attempting a masquerade or to mount a replay attack. Authentication mechanisms might include a claimant presenting the authentication information (such as a password) to the verifier, who then authenticates it. Other examples are protected exchange of authentication information, e.g. using cryptographic techniques or one-way functions, or challenge-response techniques. Either one-way or mutual authentication may be provided. In a general situation, when two parties wish to authenticate each other, they may need to involve one or more third parties.

- Access Control and Authorization: This service provides the ability to limit and control access to host systems, applications and information, and to limit the ability of resources to perform certain actions. In an access control scheme, initiators attempt to access other targets. Access control information used in the decision process includes the following: individual identities of initiators and targets, group identities of initiators and targets, security labels of initiators (clearances) and targets (classifications), roles (system administrator, manager) of initiators and targets, the actions or operations that can be performed on the target and other contextual information, which may include time periods, routing information, and location information. An access control policy essentially specifies a set of rules which define the conditions under which initiators may access targets. The decision to grant or deny a particular request is determined using access control rules and access control information associated with the request.

Traditionally, there have been two major types of access control policies: Rule-based policies and Identity-based policies. Rule based policies impose restrictions on all initiators, and are part of a mandatory access control. These controls apply to all entities, and all information. The system itself has mechanisms which can enforce elements of the security policy. A common rule-based access control policy is based on the use of security labels. It restricts access to resources based on the sensitivity of the target (classification) and the possession of corresponding access control information of the initiator (clearance).

Identity-based policies are based on individualized access control information, such as the identity or role of the initiator. These are often referred to as discretionary; they do not enforce a set security policy, but merely permit a user or administrator to prevent access to files as the owner sees fit. Note that access control policies can also be specified in terms of groups of initiators or entities acting on behalf of initiators.

A distinction often drawn between rule-based and identity-based access policies is that the former is administratively-imposed, while the latter is owner-selected. In terms of this model, the distinction lies in the control and management of access control information. A variety of choices for distribution or centralization of control is possible in a distributed environment, ranging from a completely administratively imposed policy to a completely user-selected one. Moreover, the clearances of the rule-based policy and the initiator attributes of the identity-based policies are essentially the same. The clearances can be considered as particular access control information associated with the initiator.

An access control list mechanism is convenient when a fine granularity of access control is required, and when there are few initiators. Revocation is also easier with this mechanism, as compared to others. Revocation essentially involves appropriate modification of access control lists of the target. This scheme is not suitable when the initiator population changes frequently. Capabilities are convenient when many initiators access few targets. The security label access mechanism is convenient when many initiators access many protected targets, and a coarse granularity of access control is required.

- Delegation: An important requirement that commonly arises in a cooperating environment associated with access control is that of delegation. The principle behind delegation is that one entity can authorize another entity "to act on its behalf". Delegation allows us to have a more flexible and dynamic form of access control. The security model cannot provide generic mechanisms to allow for delegation.

The originator or delegator may wish to give limited rights rather than delegate all rights. Furthermore. the delegator may only want to grant these rights for a limited period of time. The delegator should be able to identify each delegation made so that those delegation can be revoked at any time. The entity to whom the rights have been delegated must be able to determine when the rights are no longer in effect, as well as, the scope of privileges, and how long they will last. This can be done by passing a delegation token. The end point receiving a request from an entity claiming some delegated rights must be able to read, authenticate and verify the tokens specifying the rights and their duration. The end point can then make an access decision based on knowledge of the requestor's rights, using the delegation token and the access control information.

- Information Confidentiality: Confidentiality provides for the protection of information from unauthorized disclosure. The mechanisms typically used to provide confidentiality are based on cryptographic techniques. In a distributed operating system environment, it may be sufficient to protect the confidentiality of information by using access control mechanisms. In a networked environment, link-to-link or end-to-end encryption can be used.

- Information Integrity: Integrity provides for the protection of information from unauthorized modification. The modification may involve alteration, insertion, or deletion of information.

Integrity involves: generation of integrity checks (at the originating end), and verification of integrity checks (at the receiving end). Integrity mechanisms employ cryptographic techniques to produce checksums to determine whether information integrity is intact.

- Non-Repudiation: Non-repudiation provides the proof of the origin or delivery of information. This protects the sender against the threat of false denial by the recipient (proof that the information has been received) and protects the recipient against the threat of false denial by the sender (proof that the information has been sent).

The common mechanism used for providing non-repudiation relies on the use of digital signatures.

- Auditing and Accountability: Security audit is complementary to all the security services described above in that it is not directly involved in the prevention of security violations. Rather, it assists in their detection. The security audit can in turn be used to test the adequacy of security controls, and to determine whether system conforms to the security policy, and to assist in making modifications to the security policy.

Auditing services are crucial both within a system and over the networks. Following audit analysis, an entity may be held accountable for its actions. Violations or attempted violations of system security may be traced uniquely to the source. Auditing is provided by first defining the security related events, and generating security audit and/or alarms. The security audit is then analyzed to evaluate the correctness and adequacy of the security policy. For secure reporting of audit information, auditing may use other services such as authentication, integrity, confidentiality and non-repudiation.

- Availability and Prevention of Denial of Service: Denial of a service can be regarded as an extreme case of information modification, in which the information transfer is either blocked or drastically delayed.

To prevent such an attack, information can be periodically exchanged to ensure that an open path exists. The greater the frequency of exchange, the shorter the time period during which the denial of service attack will remain undetected. However, the disadvantage is that the effective bandwidth of the network is reduced.

When considering the security needs of a distributed operating system, it is necessary to understand how to manage security and how changes in the security policy and its enforcement can take place. For instance, in the case of confidentiality and integrity services, encryption and decryption keys need to be managed. In the case of access control service, access control lists and the access rules must be managed. Similarly in the case of authentication, passwords and keys, must be managed. In the case of auditing, audit trails and audit analysis is controlled and managed.

In distributed systems, it is likely that no single authority controls the entire environment. For instance, in an organization several security managers may be responsible for a subset of users, objects and operations. This does not mean that central control of security in a distributed environment is impossible. However, even with centralized control, authorities responsible for enforcing appropriate security on peripheral systems must be trusted.

6.2 Security Domains

One proposed way of managing a large distributed environment is to partition it into separate security domains. A security domain is a subset of elements that are controlled under a security policy administered by a single authority. Examples of security domain in an open systems environment include: application processes and human users.

Different relationships can exist between security domains. For instance, domains may have a peer-to-peer relationships, such as two separately owned networks. Domains may also have subdomains. In order for interactions to take place between domains, there must be common aspects in the security policies of the domains.

For secure interaction between two independent security domains to be feasible, the security policies must enable common services and mechanisms to be selected. Security attributes in each security domain must be related to each other. In the case of security subdomains within the same security domain, secure interaction between subdomains can be established by the security domain authority. Secure interaction between domains via a common third party domain can be established by the common security domain for activities within the domain's jurisdiction.

In general, when developing such a multi-domain system, the activities involved in each domain must be identified, as well as the role and functions of the authorities involved and the interaction between these authorities. Typically, additional third parties may be needed to hold some security relevant information, and to act as an arbiter in resolving conflicts.

Trusted mechanisms are essential to ensure the enforcement of security controls, and to provide assurance that the correct security measures are operating. The combination of security services and mechanisms which have been chosen to meet the security requirements and to implement the security policy mst rely on some collection of agents - whether human, software or hardware - to carry out the security functions correctly. In particular, the security management authorities mentioned above must be trusted to maintain the security of the system.

7.0 Trusted Application Development Using MLS Operating Systems

7.1 Background

Trusted systems, in particular those which can operate in multilevel or compartmented mode, have only recently become commercially available. Trusted applications, such as trusted Database Management Systems (RDBMSs), are also being developed, but they lag behind trusted operating system technology, if only because trusted operating systems serve as a platform for the trusted applications.

The design and implementation of trusted applications do not merely entail selecting arbitrary applications, placing them on a trusted operating system, and watching them operate correctly. The applications must be carefully integrated into the trusted environment provided by the operating system. For its part, the operating system must support the applications by providing certain security services or functionality. Unfortunately, little work has been done in the area of determining the security capabilities are required of trusted operating systems to support trusted applications. Security guidelines and evaluation requirements such as those found in the Trusted Computer Security Evaluation Criteria (TCSEC) mandate certain requirements on trusted operating systems. However, these requirements ensure that the trusted operating system is secure, but they do not necessarily ensure that the operating system provides the necessary security services needed to develop trusted applications.

This section of the report provides some insight and suggestions on security services, and how they could be incorporated into trusted distributed operating systems to make them a better base for developing trusted applications. The incorporation of these services would not only simplify their use in building complex distributed applications, but would also aid in reducing the cost of accreditation and development of the integrated system.

7.2 Trusted versus Untrusted Software Applications

Before discussing the operating system support for trusted applications, we need to define "trusted application" and "MLS applications". An application is a set of software components which performs some specific set of functions in order to solve a specific problem. A trusted application is one which performs some specific security task in the process of satisfying a needed operational function. A trusted application is trusted because of the operations performed by it, not because it makes use of the underlying trusted operating system.

If our definition were this broad, then every application on the system would be trusted since they all make some use of the underlying operating system. An application is considered trusted only if the application itself performs some security related function. The nature of the security performed by the trusted application can vary, but typically entails label manipulation, access control, or authentication. If an application can violate the security policy of the system, because of the context in which it executes, it must be considered trusted. This trust is not the result of performing some security-related function, but because the application has the potential to do harm such as disclosure or denial of service. In the process of analyzing a complex MLS system for trusted components, it is the latter example that is often more difficult to pinpoint and study.

In our experience, MLS application and MLS systems are systems which simultaneously permit information labeled over a range of sensitivity levels and users cleared through a range of levels to exist on the same system. For example, the system may process UNCLASSIFIED through TOP SECRET data and have users with maximal clearance levels that range from UNCLASSIFIED through TOP SECRET.

MLS applications have the potential to process data with multiple sensitivity levels for users logged on the system. In many cases, the security functionality requires these applications to keep track of the sensitivity labels of the data and to process it into comprehensible composite views of the combined information. The MLS applications may not provide any other specialized security functionality other than to manage the sensitivity labels of data contained within the process. This may include label management of information which originated from a database where the data was already labeled or newly created data which is typically labeled at the sensitivity level of the creating process. This view of MLS applications may differ somewhat from the more traditional approach to MLS systems, since in traditional views there is little labeling granularity, and an individual only processes data that exists at the sensitivity level of his process.

It is important to note that trusted applications are required only when the operational needs of the application cannot be accomplished or cannot be securely accomplished even with the support of the trusted operating system. Classic examples are the emerging trusted relational database systems that require labeling with mandatory access control at a granularity not provided by the underlying trusted operating system. To satisfy this need, a trusted RDBMS requires trusted code to enforce labeling and mandatory access control at a granularity finer than that of the operating system. If the underlying operating system performed these services, the RDBMS would not need to be trusted.

For trusted applications to perform their tasks, it may be necessary for the application to override some portion of the security policy of the underlying operating system. Typically, applications that perform trusted actions are granted operating system privileges that permit the application to override the security policy of the system. Some applications may enforce their own security policy components which are orthogonal to that of the underlying operating system, such as integrity.

Such a security policy component is independent of the policies enforced by the underlying operating system.Thus, trusted applications are required when a particular functionality needs to be provided in a secure manner, but for some reason the application cannot rely on the underlying operating system to provide the necessary security service.

7.2.1 Operating System Support for Trusted Applications

A trusted operating system to provides support for trusted applications in a variety of ways. One possible way is for the operating system to provide some security capability the trusted application depends upon to function properly. Another way is for the operating system to provide support that will help limit the possible complexity of the trusted application. If the trusted operating system provides a needed trusted service, and does so in a manner which is highly flexible, the need for the trusted application to provide the trusted capability on its own may be eliminated. The complexity of the trusted application would be limited, or may become an untrusted application, which would inevitably lessen the cost of the application, speed up its development time, and lessen the accreditation effort.

Finally, the trusted applications that implement policies that are dependent on the underlying operating system also can potentially of interfere with the operations of the underlying operating system. Such trusted applications must be carefully evaluated. Because of the possibility that the applications may interfere with the underlying trusted operating system, the operating system itself may need to be reevaluated. In addition, previously evaluated :rusted operating systems' utility may be lessened since the already evaluated operating system may require reevaluation even though no changes have been made to the system itself. Designing a distributed operating system to lessen or eliminate the need for reevaluation in such cases would advance the development of trusted applications and MLS systems.

7.2.2 Trusted Data Distribution Pathway (TDDP)

Many trusted applications need to communicate with a user or another process in a manner that clearly and unambiguously indicates the user is interacting with a trusted application, and not untrusted software. The traditional means of establishing such communication is via a trusted path. The trusted application should not provide the trusted path entirely on its own, since trusted paths often require some direct hardware interaction. Ideally, only the underlying trusted operating system should have access to the hardware.

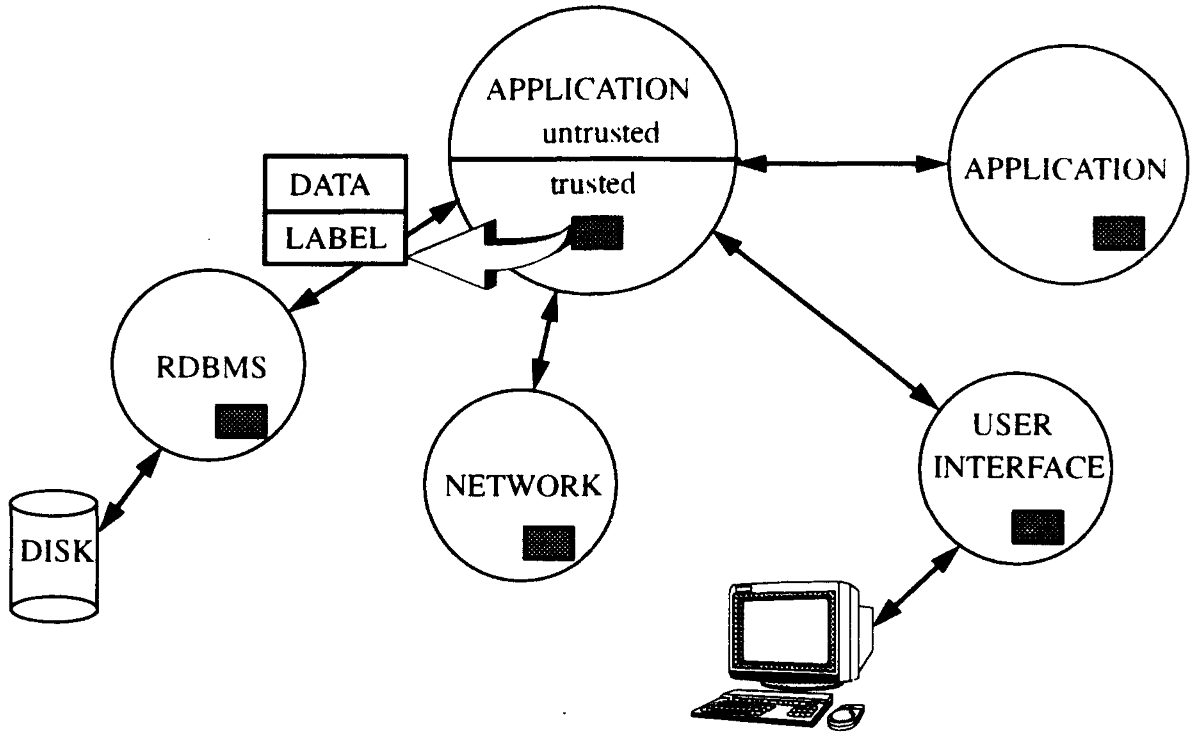
The underlying trusted operating system needs to provide services to support a trusted path mechanism to the trusted application. The trusted application could connect its trusted path mechanism with that of the underlying trusted ›erating system, to ensure an unbreachable and secured path from the user to the TCB of the operating system. Trusted path services should be bidirectional. Either the user or the trusted application should be able to use the trusted path, primarily to accommodate the highly interactive, real-time operations of many trusted applications. Consider, for example, a real-time environment where a request for information is sent to a trusted process monitoring an information source, and some action must be performed by the requesting process while it awaits an asynchronous response to its request. When the responding process does reply, the requesting process must know that the response has come from the requested source, and not from some untrusted code that is attempting to spoof. When the responding process is able to invoke the trusted path, the user can be assured that the communication did indeed come from the trusted process.

The development of MLS applications would be simplified by providing trusted services which support trusted data distribution pathways. To enhance development of real MLS systems, not only must trusted data distribution pathway components be provided which support *inter*process communication, some sort of *intra*process trusted data distribution pathway mechanism is also necessary.

Figure 3 illustrates these issues. The figure shows several processes which comprise a trusted data distribution pathway for moving labeled data structures between the database or network interface, a MLS application, and the user interface process which displays data on the user's terminal. Processing of data may also include communication to another MLS application as well. In order for this group of processes that comprise a single logical MLS system to act in concert and provide trusted labeling of data, they must all enforce the same security policy and adequately protect the labeled data object present within the virtual memory of each process. If the labeled data object is compromised at any point in the pathway then the label associated with a data element is not trusted. If the underlying operating system supported the means to construct a trusted data distribution pathway, then these processes could use the mechanism to implement a consistent and safe transfer of labeled data throughout the system. This would minimize the amount of trusted code within each process and ensure consistent treatment of the labeled data.

The figure also illustrates the issue of *intra*process protection of labeled data structures. The MLS application process in the center of the pathway is large and complex. In order to minimize the evaluation of this process, the labeled data structure contained within the process must be securely segregated from the untrusted code components. Otherwise, all components of the process could access the data structure and violate security policy. An MLS operating system could simplify this problem by providing trusted memory management functions which enforce domain separation within a single process. If this were the case, then only those portions of the process which can access the protected data structures would need to be evaluated.

Figure 3:Trusted Data Distribution with *Inter*process and *Intra*process Components



As we found in the MLS/GDSS project, one of the major obstacles was the segmentation within a process of trusted and untrusted data structures. Many current development efforts fail to recognize this issue, and find that they must accredit very large amounts of application code. In most cases, this code was not developed entirely to trusted code development standards. Rather, small portions recognized as performing trusted operations were developed under rigorous standards, and then mixed into the remaining code which performs other non-security related functions. Within a single operating system process, no domain separation exists which permits the protection of critical data structures.

For example, suppose a trusted application is needed which reads data and associated sensitivity labels from a trusted data repository, and then writes that information to a file or printer. Such a process might form part of a trusted reporting utility providing some granularity of labeling beyond that of just the file or the operating system process. Ideally, the data structures contained within the virtual memory of the process that hold data values and their associated sensitivity labels must be protected from all but the trusted code segments within that process. If not, it would be necessary to treat every code segment comprising the process as trusted, since the critical data structures are vulnerable to modification by any code executing within the process. Development of true MLS applications would be greatly simplified, if existing MLS operating systems provided *intra*process memory management capabilities that permitted domain separation of untrusted versus trusted data structures and code.

In order to provide such services, extending the privilege granularity from that of the process may be possible. If the MLS application developer could create protected memory structures, which were managed by trusted operating systems mechanisms, and were accessible only by routines within the process which were predesignated to violate the operating systems' protection of memory structures, then the *intra*process domain separation problem would be significantly reduced. If an MLS system were written in such an environment, evaluation and accreditation would only include data structures and code segments that were deemed trusted by the developers.

These trusted development services could be made available by the operating system to the trusted application via system calls or library routines. For additional control, only appropriately privileged trusted applications could be allowed to utilize the system calls or library routines. Interestingly, none of these proposals - use of library routine or systems calls, use of privileges, bidirectional trusted data distribution paths - is beyond the current state-of-the-art. None of these proposed mechanisms conflict with the requirements of the various security evaluation documents such as the TCSEC. Unfortunately, because these capabilities are generally not called for in the various security guidelines. Unfortunately, none of the trusted operating systems provide these capabilities.

7.2.3 Privileged Processes and Privileged Code Segments Support

As noted earlier, in order to perform the necessary functions, many trusted applications require some privileges from the underlying operating system. Increasing privilege granularity and supporting privileged code segments improve support for the development of trusted applications and the MLS system. The use of fine granularity of privilege is consistent with the TCSEC concept of Least Privilege, and aids the evaluation of trusted applications. In addition, use of fine-grained privileges and privilege assignment may help minimize the reevaluation of the underlying operating system. If a system only supports a single "super user" privilege which can override operating system policy, then the application only requiring the ability to write to the operating system audit trail will unnecessarily be given the ability to override the mandatory and discretionary access control policies, as well as the system audit policy. Such action clearly violates the concept of least privilege. The entire operating system must also be reexamined to ensure that it works properly in conjunction with the trusted application. But if the underlying operating system supports a GENERATE_AUDIT_RECORD privilege, then only that part of the operating system responsible for enforcement of audit policy would need to be reexamined. Even in trusted operating systems affording finer layers of granularity, such as Secureware, the breakdown is insufficient. In most cases, i the ALLOWMACACCESS privilege must be enabled in order to violate the MAC of the operating system to perform most security related operations.

In most trusted application implementations, large trusted applications execute within a single process. As described earlier, this often necessitates the trusting and evaluation of all code comprising the process, since security domain support is not provided by trusted operating systems. Another approach is to segregate the untrusted application code from the trusted application code using separate processes. The trusted operating system provides the domain separation, through process separation mechanisms, ensuring that untrusted code in one process can not manipulate trusted data structures in another process. This is not a feature unique to trusted operating systems since all operating systems guarantee process independence. This approach also has its limitations. Using multiple processes requires that secure interprocess communication mechanisms be available to the trusted application programmer. The development of the trusted applications is complicated by this approach. The communication between the processes must be carefully evaluated to ensure that the domain separation is not violated by a broadly defined interface and that only certain specifically-identified processes can communicate with each other. Since the trusted application is now split amongst two or more processes, additional complications may arise. For example, if both processes, the untrusted component and the trusted component, need to communicate externally to a RDBMS or network socket, they act as independent agents. The two processes may compete for information or locks, although they are trying to coordinate their actions.

In addition to finer granularity of privileges, a finer granularity of privilege assignment is also needed to better support the development of trusted applications and MLS systems. Such control might be assignment of privileges at the routine level within a library. Using privilege bracketing to support domain separation within individual processes also requires support for trusted memory management services provided by the underlying operating system. For example, it should be possible to allocate a trusted memory segment which is managed and protected by the operation system. From within the allocating process, this memory segment would be created only by designated routines and be readable or writable only by routines with the properly designated privileges. One might envision a CREATE_PROTECTED_MEMORY and a READ_PROTECTED_MEMORY privilege, both of which are assignable to individual routines. The trusted operating system would need to provide security services that could be called to create and read/write the protected memory (for example, a protected_malloc() call). This increased granularity of privileges and privilege assignment would provide several desirable benefits for the development of trusted applications in distributed MLS environments:

- domain separation between untrusted and trusted data and code segments would be possible within a single large process, simplifying the development of trusted applications. Use of multiple processes to provide this functionality would be unnecessary.
- support for the trusted data distribution pathway through the application would be available.
- the security accreditation would be simplified, since evaluation could focus on the routines assigned privileges.

It is difficult to define the appropriate set of privileges t a trusted operating system should enforce without adversely affecting the flexibility of the operating system. However, wherever possible, the operating system should enforce as fine a granularity of privilege as necessary. The set of privileges provided by a trusted operating system should take into account likely requirements of trusted applications. The choice of privileges should also take into account the requirements against which the operating system will be evaluated. At a minimum, privileges should not cross policy boundaries. Within a given policy, capabilities that map to different requirements should not couple together in a single privilege. Such security services, if implemented, would not only ensure that privileges conform to the concept of least privilege and aid in the development of trusted applications, but in so doing would help minimize the amount of reevaluation required of a trusted operating system.

When incorporating a trusted application into an already evaluated trusted operating system, the trusted application has the potential to interfere with the operations of the trusted operating system by applying its privileges to resources under the operating system control. For this reason, the addition of a trusted application to an already evaluated operating system requires that some of the underlying operating system must be reevaluated. The use of fine-grained privileges and privilege assignment limit the amount of the operating system reevaluation. Encapsulating the actions of the trusted application will further limit the amount of reevaluation necessary.

Encapsulation could be achieved by enforcement of an object typing policy by the trusted operating system. Under such a system, the operating system would ensure that all applications, both trusted and untrusted, would have some type associated with them. The object typing policy would ensure that an application of one type could only access subjects and objects of some specified type. Some systems already support such capabilities (LOCK, XTS-UX). The most restrictive variation of this policy would ensure that an application could only access subjects and objects of the same type. By imposing such restrictions, one could ensure that a trusted application could not interfere with the actions of the trusted operating system or some other trusted application.

An application encapsulation mechanism would constrain which subjects and objects a trusted application could access, and, therefore, would ensure that trusted applications could not interfere with the workings of trusted operating system components. This satisfies the security and domain separation requirements of higher assurance trusted operating systems, but could cause some operational difficulties. Some entities of one type occasionally need to be accessed by entities of another type. But if the encapsulation policy is such that an application can only access entities of the same type, trusted application' will be prevented from accessing the object of a different type. An alternative is to enforce a less rigid policy which would allow certain objects to be accessed by subjects of specified types or permit specified objects to access objects of multiple types. These alternatives weaken the encapsulation policy, however.

From an implementation perspective, an object oriented trusted operating system could most easily implement such functionality. The trusted operating system would provide services for creating objects of certain types and specifying how and by what subject types these objects could be accessed. This approach would support the domain separation of trusted applications and the underlying operating system minimizing the accreditation effort.

7.2.4 Labeling Granularity and Security Policy Consistency

Trusted application were primarily developed to enforce a finer granularity of labeling and mandatory access control than that enforced by the operating system. We found this to be a strong necessity during our work on the MLS/GDSS project. In our experience, users require trusted sensitivity labels to be available at the single data element level. The labeling granularity provided by the operating systems (e.g., file level) is generally insufficient to satisfy the needs of most trusted applications. Some architectures have attempted to address this problem by aligning the objects of the trusted application with that of the operating system. Thus, in the case of an RDBMS, all of the Secret tuples of a relation would be in a Secret file and all of the Top Secret tuples would be in a Top Secret file. For many environments this is an acceptable solution. However, for systems which require an extremely fine level of granularity, and a large number of different security levels, this architectural approach is unacceptable. A side effect of this approach is a severe degradation in performance, since logically related objects may exist in multiple files. RDBMS performance, for example, degrades logarithmically with the number of security sensitivity labels.

For the performance reasons cited above, many trusted applications often provide their own mandatory access control and their own label manipulation and label conversion routines. For most systems, these label and mandatory access control are small and simply to understand. However, in other systems, these label routines may be complicated. Duplication of labeling and mandatory access control routines in the application can prove to be expensive. In such environments, labeling and MAC routines should be available to the trusted application. The code could be placed in library routines or the appropriate system calls could be provided. Making such services available to the trusted application reduces burden, making the applications less complicated, less expensive to build, and to accredit. In addition, such an approach also ensures that the applications and the operating system are using labels consistently, thus eliminating the need to convert labels when data is passed between the operating system components, the network and the trusted applications.

A trusted operating system can also support some of the labeling and MAC requirements of trusted applications by providing an arbitrarily fine level of labeling granularity down to the byte or to the user-defined data structure level. By having an operating system provide such fine labeling granularity, the trusted application no longer needs to provide its own labeling, but can rely instead on the operating system.

One of the nice features which would result from better operating system label management would be support for true MLS files, and possibly, a MLS file system. One of the current drawbacks of the existing MLS systems is that they are only capable of labeling at the file level. If an operating system could provide a programmer selectable labeling granularity, application objects would not need to be physically stored in separate operating system files.

7.2.5 MLS Memory Management

Within a trusted application, it is often necessary to control the labeling of critical data structures according to a predetermined security policy. The security policy might state, for example, that any modification to a data structure results in the relabeling of any data within that structure to the sensitivity level of the executing process. The trusted operating system would then provide the following abilities:

- create protected memory structures, using trusted memory management routines.
- provide labeling services which allowed the trusted application to designate a data structure as a labeled object whose sensitivity label is managed by the trusted operating system.
- provide labeling services that permit the reading and writing of the label of a designated object from appropriately privilege code segments.

The lack of physical partitioning means that system performance would not be impacted by the number of sensitivity levels, as is the case with most trusted operating system today. Because the trusted application would not need to enforce its own labeling or mandatory access control, the cost and complexity of the application would be reduced, and the certification of the application would be simplified.

7.2.6 MLS Message Passing

A natural extension of enhanced labeling services and label management by the trusted operating system is support for true MLS message passing. These messages could be files or byte streams originating from a network socket or an interprocess port. An operating system supporting true MLS message passing capabilities would permit MLS data streams to be sent between processes at different sensitivity levels and ensure that only the properly labeled data is transferred between the processes. For example, if an MLS file containing a byte stream with an enhanced labeling granularity (such as a user defined record boundary) is sent from a TOP SECRET process to a SECRET process, only SECRET data would be transferred to the SECRET process. This operation would require no privileges. If the TOP SECRET process needed to downgrade and send the TOP SECRET data contained in the file or byte stream, then the process would require some sort of DOWNGRADE privilege. In today's trusted operating systems, in order for a TOP SECRET process to communicate with a SECRET process, trusted code and privileges are required regardless of whether TOP SECRET data is transferred between the processes. With enhanced label granularity and true MLS message passing capabilities provided by a trusted operating system, much of the need for trusted applications disappears. By removing the need for privileges from the applications, we greatly reduce the cost of development and accreditation of MLS systems. These sorts of capabilities would also simplify the development of distributed applications.

7.2.7 MLS Processes

Another capability that would enhance the development of MLS applications is support for true MLS processes. MLS processes are processes that can change there sensitivity label during execution. Presently, no MLS operating system provide such support; processes remain at the sensitivity level at which they were created. To minimize the amounts of trusted code in an application using today's MLS operating systems, a single privileged dispatcher program is written which creates of forks child processes at levels other than the execution level of the parent. Unfortunately, this complicates problems of resource contention, lock conflicts, and data sharing through interprocess communication.

Often times it is necessary for a single process to manage data originating from multiple sources of differing sensitivity levels. If the trusted operating system supported MLS processes, then a single process could securely manage MLS data and communicate with other single level processes. This communication would not require privileges, such as DOWNGRADE, but would be accomplished through the process changing its sensitivity level to match that of the other process. Obviously, any data structures whose sensitivity level is not dominated by the new sensitivity level of the process would become inaccessible to the process until it returns to an appropriate sensitivity level. Support for MLS applications would reduce the amount of trusted components on and MLS system and simplify the architecture of the systems by allowing a single MLS process to manage MLS data, rather than a collection of single level processes communicating with each other.

8.0 Informal Requirements for a Distributed MLS Operating System

8.1 Comparison of Existing Operating Systems

Table 1 presents a rudimentary comparison of the operating systems discussed in this report. The features that are listed include the following:

- Microkernel/Monolithic - This indicates if the operating system is based on a microkernel design or a monolithic kernel design. This distinction requires some caution since monolithic operating systems such as PLAN 9 are actually smaller in size then microkernel-based operating systems such as MACH.
- Distributed - This indicates if the operating system is distributed.
- Heterogeneous/Homogeneous - This indicates for distributed operating systems if they are supported in an environment comprised of different types of processors. Many of the distributed operating systems function on different hardware platforms but are not truly heterogeneous.
- Process migration - This indicates if the operating system supports automated process migration between different processors in a distributed environment.
- MLS - This indicates by assurance level the evaluated or targeted level of the operating systems providing MLS capabilities.
- Real-time - This indicates if the operating system provides real-time processing capabilities. No distinction is made between the various real-time algorithms supported by the various real-time operating systems.

Table 1: Comparison of Several Existing Operating Systems

FEATURE	MACH	SPRITE	AMOEBA	PLAN 9	MARUTI	ARTS	RT-MACH	CHAOS	CX/RT (/SX)	CMW+	TMACH
Microkernel/ Monolithic kernel	micro	mono	mono	mono	mono	mono	micro	micro	mono	mono	micro
Distributed	+	+	+	+	+	+	+	+	-	-	+
Hetergeneous/ Homogeneous	-	-	-	+	-	-	-	-	-	-	-
Process migration	-	+	-	-	-	-	-	-	-	-	-
Multilevel secure	-	-	-	-	-	-	-	-	B1/B2	B1+	B3
Real-time support	-	-	-	-	+	+	+	+	+	-	-

8.2 Problems with Existing MLS Operating Systems

In virtually every case, MLS capabilities have been added to existing operating systems. Certainly the most popular MLS systems are based on existing non-MLS operating systems, to which MLS capabilities have been retrofitted. Most of the current MLS operating systems which have undergone formal evaluation at the NCSC are designed to meet B1 assurance levels. There are several other UNIX-based MLS operating systems that exist or are under evaluation which meet higher B2 and B3 levels of assurance, such as System V 4.2 ESMP or Trusted Mach.

Unfortunately, retrofitting MLS functionality onto existing operating systems has resulted in a number of drawbacks. The MLS versions of existing operating systems tend to have a limited assurance level, typically B1. Most of the B1 level MLS operating systems are based on monolithic UNIX operating systems or similar monolithic structures such as VMS. The architectural structure and size of these operating systems makes it difficult to meet the security requirements necessary to reach higher levels of assurance primarily because of a lack of domain separation and shear size. Unfortunately for most MLS environments, a B1 level of assurance is insufficient to meet operational requirements. The goal of most MLS systems is to process UNCLASSIFIED through TOP SECRET data with some users only cleared through TOP SECRET. These scenarios require at least a B3 level of assurance.

The MLS version of existing operating systems have limited support for development of MLS applications or worse, restrict development of single-level applications. Most MLS operating systems implement a limited Bell-LaPadula security model. In many systems, write up operations are not permitted without privileges, even though this is supported by the Bell-LaPadula model. In these systems, the file system has been modified to include security classification labels but not to provide true MLS capabilities. Process management also includes security classification levels, but again with severely limited MLS capabilities.

MLS application development and certification of these systems is complex and costly because the systems lack true MLS capabilities. In many cases, it becomes necessary to develop large trusted components to overcome the lack of MLS capabilities in these systems or to circumvent the restrictions imposed by the system to meet security evaluation. In many MLS systems, functionality that was removed from the operating system to meet formal evaluation criteria is placed in trusted application which must undergo certification before the system can be utilized. Because certification tends to be less strenuous than formal evaluation by the NCSC, what often results is a system that is less secure than if the applications were removed from the system.

8.3 Design for Distributed Real-Time MLS Operating System

To move forward with MLS systems, it is necessary to develop an distributed real-time MLS operating system designed to provioe true MLS capabilities. This development would not encompass the retrofitting of security services to an existing operating system as in the past. Instead, a MLS operating system is needed, one which is designed from its inception to provide a high level of assurance and a sophisticated MLS application development environment.

Such a system would include the following functional requirements.

8.3.1 Distributed MLS Message Passing

Support for MLS message passing would improve support for development of MLS applications. MLS message passing provides core capabilities to build sophisticated MLS services such as Remote Procedure Calls and I/O streams in a trusted MLS operating system. Such functionality would permit the exchange of MLS information between multi-level processes in a distributed environment. A distributed micro-kernel design of the operating system is best suited to construct MLS message handling services and to reach a B3 level of assurance because of the need to minimize the trusted code components and size of the kernel and to provide adequate domain separation. MLS message handling capabilities would greatly enhance the development of trusted applications by reducing the amount of application specific trusted code. A MLS message handling capability would permit MLS I/O streams which could handle single byte streams containing data at different sensitivity levels. This functionality would further simplify development of a distributed MLS operating system with trusted network communications.

8.3.2 MLS Memory Management

The second critical component of a distributed MLS operating system is support for MLS memory management. This feature, in conjunction with MLS message handling, would provide extensive support for the development of MLS applications and reduce the cost and complexity of these systems by minimizing the trusted components and the accreditation effort. MLS memory management support would require finer granularity of privileges than those found in existing MLS systems and the ability to assign privileges to code segments within a single process. Enhanced labeling granularity through user definable labeled data objects to support MLS applications, graphical environments, and trusted data distribution would also be provided as part of the MLS memory management services. This capability would significantly enhance the development of MLS application that are distributed.

8.3.3 MLS File System

MLS file systems to support distributed file access in an MLS environment would also enhance development of MLS systems. Much complexity and need for trusted components would be eliminated if the underlying operating system provided a true MLS file system. No MLS operating system to date transparently supports this capability.

8.3.4 Real-Time Support

Support for real-time activities in distributed environments is well documented and has been implemented in several systems. Support for real-time services does not impose any additional constraints on a trusted operating system than it does on an untrusted distributed operating system. If the problems of supporting development of MLS applications in a distributed trusted environment are solved, addition of real-time capabilities should not be difficult. At least one evaluated B1 operating system (CX/SX) already provides real-time capabilities and a number of untrusted distributed operating systems exist which support real-time operations. If the complexity of developing MLS applications could be reduced through support for MLS message handling and MLS memory management and associated functionality, then development of real-time applications with their unique timing constraints would also be helped.

8.3.5 Process Migration

Process migration capabilities would permit redistribution of tasks during changing system loads and improve computing resource utilization in a distributed environment. A limited number of the distributed operating systems, such as Sprite, successfully provide this functionality. Process migration in a homogeneous distributed environment is relatively simple to solve. However, in a heterogeneous distributed environment, this problem becomes more complex because of translation of data structures between differing hardware platforms.

Even without process migration capabilities, implementation of a distributed heterogeneous operating system is complex. The operating system environment must provide support to activate different binary versions of a a program depending on the hardware platform. Some systems, such as PLAN 9 have implemented rudimentary solutions to this problem.

8.3.6 Uniform Name Space

A uniform and transparent name space is needed to support a distributed file system. Several of the distributed systems have solved this problem in different means. Use of dedicated but distributed file servers to enhance performance like those found in Plan 9 is especially appealing.

8.3.7 Distributed Security Policy

A security policy which is distributed and consistent throughout the system is also needed to support distributed MLS systems. No trusted distributed operating system has fully implemented this feature. However, TMach is architected to provide this functionality, once distributed capabilities are added to the system. Support for MLS memory management and granular privilege assignment would simplify implementation of a distributed security policy since the various processes which comprise the distributed environment would rely on MLS services provided by the trusted operating system. The need for individual components of a distributed MLS application to implement its own security services would be eliminated.

9.0 Summary

This report has discussed general and specific issues of distributed, real-time, and trusted operating systems. The report is meant to provide an introduction to some of the issues, and not a detailed comprehensive discussion of each of these topics. In analyzing a number of existing operating systems, it is obvious that no single system exists that meets all of the goals of a trusted distributed operating system. In some cases, the functional needs are in direct opposition to each other. An even greater concern that exists is the relatively poor support for the development of trusted and MLS applications in current trusted operating systems. We believe that this is mainly the result of the relatively restricted ways in which MLS operating systems have been developed from preexisting standard operating systems. In order to overcome many of the problems associated with distributed operating systems in an MLS environment, we suggest that development of a trusted operating system that supports construction of trusted and MLS applications be explored. With the proper security architecture and security services, a highly usable distributed MLS operating system can be constructed using current technology.

10.0 References

1. Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 Usenix Conference*, pages 93-112.

2. R. Baron, D. Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rashid, A. Tevanian, and M. Young. *Mach Kernel Interface Manual*. School of Computer Science, Carnegie Mellon University, August 1990.

3. R. Rashid. From rig to accent to mach: The evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*, pages 1128-37, November 1986.

4. R. Rashid, R. Baron, A Forin, D. Golub, M. Jones, D. Julin, D. Orr and R. Sanzi. Mach: A foundation for open systems. In *Proceedings of the 2nd Workshop on Workstation Operating Systems, IEEE*, pages 109-13, September 1989.

5. A. Tanenbaum and R. Van Renesse. Distributed operating systems. *Computing Surveys*, 17(4):419-470, December 1985.

6. A. Tevanian, R. Rashid, D. Golub, D. Black, E. Cooper, and M. Young. Mach threads and the unix kernel: The battle for control. In *Proceedings of the Summer 1987 USENIX Conference*, pages 185-97, June 1987.

7. J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Usenix 1990 Summer Conference*, pages 247-56, June 1990.

8. M. Nelson, B. Welch, & J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134-54, February 1988.

9. S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, & H. van Straveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44-53, May 1990.

10. A. S. Tanenbaum, R. van Renesse, H. van Straveren, G. Sharp, S. Mullender, A. Jansen & G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46-63, December 1990.

11. A. S. Tanenbaum & R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419-70, December 1985.

12. R. Pike, D. Presotto, K. Thompson, & H. Trickey. Plan 9 from Bell Labs. In *UKUUG Summer 1990 Conference Proceedings*, pages 1-9, London, England, July 1990.
O. Gudmundsson, D. Mosse, A. Agrawala, and S. Tripathi. Maruti: A hard real-time operating system. In *Second IEEE Workshop on Experimental Distributed Systems*, pages 29-34, October 1990.

13. E. P. Markatos. Multiprocessor synchronization primitives with priorities. In *Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pages 1-7, May 1991.

14. X. Yuan and A. Agrawala. A decomposition approach to nonpreemptive real-time scheduling. Technical Report CS-TR-2345, Department of Computer Science, University of Maryland, November 1989.

15. H. Tokuda, C. Mercer, and J. Lehoczky. Scheduling theory and practice in arts. Technical report, School of Computer Science, Carnegie Mellon University, August 1990.

16. H. Tokuda and M. Kotera. A real-time tool set for the ARTS kernel. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1988.

17. Ahmed Gheith, Prabha Gopinath, Karsten Schwan, and Peter Wiley. Chaos and chaos-art: Extensions to an object-based kernel. In *IEEE Computer Society Fifth Workshop on Real-Time Operating Systems*, Washington, D.C. IEEE, April 1988.

18. Ahmed Gheith and Karsten Schwan. Chaos-art: Kernel support for atomic transactions in real-time applications. In *Nineteenth International Symposium on Fault-Tolerant Computing*, Chicago IL, Pages 462-69, June 1989.

19. Ahmed Gheith and Karsten Schwan. Chaos-art - kernel support for multi-weight objects, invocations, and atomicity in real-time applications. ACM Transactions on Computer Systems, 11(1):33-72, April 1993.

20. Harris Computer Corporation, *CX/UX System Adminstration Manual*, 1992.

21. Harris Computer Corporation, *CX/UX Adminstrator's Reference Manual*, 1992.

22. Harris Computer Corporation, *CX/RT Reference Manual*, 1992.

23. H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *Proceedings of teh USENIX 1990 Mach Workshop*, October 1990.

24. Hideyuki Tokuda, Tatsuo Nakajima, and Prithvi Rao. Real-time mach: Towards predictable real-time systems. In *Proceedings of the USENIX 1990 Mach Workshop*, October 1990.

25. Secureware, Inc., *CMW+ for Open Desktop Trusted Facility Manual*, 1992.

26. Secureware, Inc., *CMW+ for Open Desktop Security Features User Guide*, 1992.

27. Department of Defense Trusted Computer System Evaluation Criteria. Technical Report DOD 5200.28-STD, DoD, December 1985.

28. Information Technology Security Evaluation Criteria. Technical Report 1.1, Department of Trade and Industry, January 1991.

29. Trusted Mach Philosophy of Protection. Technical Report TIS TMACH Edoc-0003-93B, Trusted Information Systems, Inc., April 1993.

30. Trusted Mach Class Library Executive Summary. Technical Report TIS TMACH Ddoc-

0004-93A, Trusted Information Systems, Inc., December 1991.

31. Trusted Mach Root Name Server Executive Summary. Technical Report TIS TMACH Ddoc-0001-93A, Trusted Information Systems, Inc., July 1992.

32. Trusted Mach System Architecture. Technical Report TIS TMACH Edoc-0001-93B, Trusted Information Systems, Inc., May 1993.