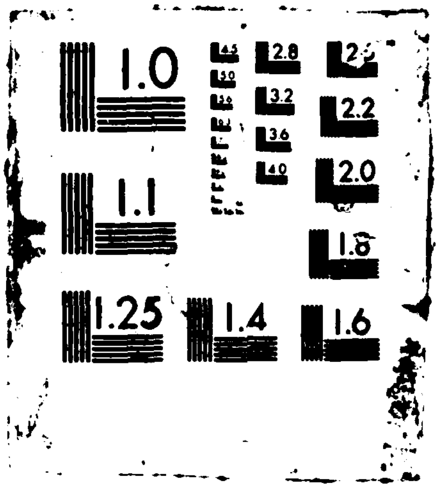
AD-A178 437 ADA (TRADE NAME) FOUNDATION TECHNOLOGY VOLUME 6 SOFTWARE REQUIREMENTS FOR..(U) INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA J STANKOVICK ET AL. DEC 86 1/1

UNCLASSIFIED IDA-P-1893-VOL-6 IDA/HQ-86-30823 F/G 9/2 NL


1.0
2.8
3.2
3.6
4.0
2.2
2.0
1.1
1.8
1.25
1.4
1.6

AD-E500 848

(2)

IDA PAPER P-1893

Ada* FOUNDATION TECHNOLOGY

Volume VI: Software Requirements for WIS Operating System Prototypes

Jack Stankovick, *Task Force Chairman*
Clyde Roby, *IDA Task Force Manager*
David Cheriton
Mike Liu
Alan Smith
John Salasin, *Program Manager*

December 1986

Prepared for
Office of the Under Secretary of Defense for Research and Engineering

IDA
INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

*Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

IDA Log No. HQ 86-30823

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA Paper does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

This paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and sound analytical methodology and that the conclusions stem from the methodology.

Approved for public release, distribution unlimited.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

AD-A178 437

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) P-1893 - Volume VI	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION WIS Joint Program Management Office 8b OFFICE SYMBOL (if applicable) WIS/JPMO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) 7798 Old Springfield Road McLean, VA 22102	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-W5-206 / WORK UNIT ACCESSION NO.

11 TITLE (Include Security Classification)
Ada Foundation Technology: Volume VI - Software Requirements for WIS Operating System Prototypes

12 PERSONAL AUTHOR(S)
J. Stankovick, C. Roby, D. Cheriton, M. Liu, A. Smith

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM ______ TO ______	1986 December	90

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
World Wide Military Command and Control System (WWMCCS), WWMCCS Information System (WIS), operating systems, automatic data processing (ADP), local area network (LAN), command, control, and communications (C3), Ada programming language, CAIS, Inter-Process Communication (IPC).

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The most important ingredient for a successful WIS is the design and implementation of a suitable distributed operating system. The WIS operating system (OS) is a distributed operating system in the sense that it provides an abstraction of a single system across network connected multiple machines. The design of WIS OS presented is a well-balanced design that has significant potential for meeting the requirements of WIS. For example, effective performance is achieved by providing a minimal kernel that optimizes local area network (LAN) Inter-Process Communication (IPC), contains a very fast context switch and supports "lightweight" kernel tasks. Security is supported in the kernel by having clearly delineated address spaces, basic mandatory access control and all communication controlled via the IPC mechanism which can ensure that the proper security access is followed.

This volume is the sixth of a nine-volume set describing projects which are planned for prototype foundation technologies for WIS using the Ada programming language. The other volumes include command language; software design, description, and analysis tools; text processing; database management system; planning and optimization tools; graphics; and network protocols.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ■ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code) / 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

IDA PAPER P-1893

Ada™ FOUNDATION TECHNOLOGY

Volume VI: Software Requirements for WIS Operating System Prototypes

Jack Stankovick, *Task Force Chairman*
Clyde Roby, *IDA Task Force Manager*
David Cheriton
Mike Liu
Alan Smith
John Salasin, *Program Manager*

December 1986

IDA

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-W5-206

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

1.0 INTRODUCTION

The World Wide Military Command and Control System (WWMCCS) is an arrangement of personnel, equipment (including automatic data processing (ADP) equipment and software), communications, facilities, and procedures employed in planning, directing, coordinating, and controlling the operational activities of U.S. Military forces.

The WWMCCS Information System (WIS) is responsible for the modernization of WWMCCS ADP system capabilities, including information reporting systems, procedures, databases and files, terminals and displays, communications (or communications interfaces), and ADP hardware and software. The WIS environment is a complex one consisting of many local area networks connected via long distance networks. The networks will contain a wide variety or hardware and software and will continue to evolve over many years.

The main functional requirements for WIS are presented in [JACK 84]. Briefly, the functional requirements have been categorized into seven areas.

a. Threat identification and assessment functions involve identifying and describing threats to U.S. interests.

b. Resource allocation capabilities must be provided at the national, theater, and supporting levels.

c. Aggregate planning capabilities must provide improved capabilities for developing suitable and feasible courses of action based on aggregated or summary information.

d. Detailed planning capabilities must provide improved methods for designating specific units and associated sustainment requirements in operating plans and for detailing the sustainment requirements in supporting plans.

e. Capabilities must must be provided to determine readiness, for directing mobilization, deployment and sustainment at the Joint Chiefs of Staff and supported command levels, and for promulgating and reporting execution and operation orders.

f. Monitoring capabilities of the system must provide the information needed to relate political-military situations to national security objectives and, to the status of intelligence, operations, logistics, manpower, and C3 situations.

g. Simulation and analysis capabilities must include improved versions of deterministic models that are comparable to those contained in the WWMCCS.

In order to support these high level objectives, the WIS system software must provide an efficient, extensible and reliable base upon which to build this functionality. To develop such system software, several projects are planned for prototype foundation technologies for WIS using the Ada programming language. The purpose for developing these prototypes is to produce software components that:

a. Demonstrate the functionality required by WIS.

b. Use the Ada programming language to provide maximum possible portability, reliability, and maintainability consistent with efficient operation.

c. Display consistency with current and "in-progress" software standards.

Foundation areas in which prototypes will be developed include:

a. Command Language

b. Software Design, Description, Analysis Tools

c. Text Processing

d. Database Management System

e. Operating Systems

f. Planning and Optimization Tools

g. Graphics

h. Network Protocol

1.1 Purpose

The most important ingredient for a successful WIS is the design and implementation of a suitable distributed operating system. The WIS operating system (OS) is a distributed operating system in the sense that it provides an abstraction of a single system across network connected multiple machines. The design of WIS OS presented is a well-balanced design that has significant potential for meeting the requirements of WIS. For example, effective performance is achieved by providing a minimal kernel that optimizes local area network (LAN) Inter-Process Communication (IPC), contains a very fast context switch and supports "lightweight" kernel tasks. Security is supported in the kernel by having clearly delineated address spaces, basic mandatory access control and all communication controlled via the IPC mechanism which can ensure that the proper security access is followed.

Security is also supported outside the kernel by (1) "alias" processes which implement and serve as safeguards for inter-cluster communications, and by (2) an authentication agent. Fault tolerance is provided, in part, by the distributed nature of the system, as well as by the fault tolerant distributed file system. Extensibility is enhanced because of the multi-level and modular design of WIS OS as well as the use of the Ada programming language and adherence to the structure and modularity of WIS OS can be explained by considering three main levels together with the concept of an agent. An agent is a module that implements one or more Ada packages to provide some service such as authentication, logging and auditing, or secondary storage management. The three levels are:

a. Level 1 (Kernel)

b. Level 2 (Run-Time Support)

c. Level 3 (Application)

Level 1 (Kemal) provides an efficient base for transparent (network-wide) IPC, security, and basic process, main memory and device support. It also provides basic support for the CAIS and the concept of an object. Each type of object is viewed as an abstract data type.

WIS OS objects include open files, atomic transactions, jobs, processes and virtual spaces. The kernel provides operations for invoking operations on objects between processes, such as found in the message passing scheme. It also provides operations for changing the amount of valid memory associated with a task plus mapping portions of files in and out of the address space. Other type of objects and operations on them can be defined at the other levels. Further, language processors are free to define other types of objects either using these basic WIS OS objects or independently.

Level 2 (Run-Time Support) comprises facilities that need not be implemented in the kernel and are implemented in server processes that execute outside the kernel as well as by so-called run-time procedures that execute in the address space of the invoker. The run-time support level provides this necessary OS functionality that is not included in the kernel. This level is extensible and initially includes the following agents:

a. Secondary Storage Memory Management

b. Transaction Manager

c. Program Execution Module

d. Authentication

e. Time Synchronization

f. Command Language Interfaces

g. IPC Support

h. Logging and Auditing

i. Cluster Gateways

j. Print Server

k. Multi-Window Server

l. Input/Output (I/O) Drivers

m. Name Server

Level 3 (Application) includes application programs an other agents which are not necessary for run-time support. This includes the database management system (DBMS), user application tasks, and non-essential OS utilities.

1.2 Scope

In this document the basic design, structure and interfaces of the WIS OS are provided. Since the kernel is the single most critical element for a successful system, this entity is described in the most detail. Also described in a lot of detail are the following run-time support agents: Secondary Storage Memory Management, Program Execution Module, Transaction Manager, Authentication Server, IPC Support, the Cluster Gateways, and the Multi-Window Server. Areas included in this document but not discussed in detail include: Time Synchronization, Command Language Interfaces, Print Server, and I/O Drivers. The Logging and Auditing Module and the Name Server are not discussed in this document.

Most of the WIS OS will be implemented in the Ada programming language and adhere to. Finally, this design is based on a variety of distributed system projects including V [CHER 84], Cronus [SCHA 85], Accent [RASH 81] and Cedar/Mesa.

1.3 Terms and Abbreviations

ADP	Automatic Data Processing
ADT	Abstract Data Type
APSE	Ada Programming Support Environment
CAIS	Common APSE Interface Set
CL	Command Language
DBMS	Database Management System
DES	Data Encryption Standard
DS	Disk Server
FCFS	First Come First Serve
FS	File Server
GKS	Graphical Kernal System
I/O	Input/Output
IP	Internel Protocol
IPC	Inter-Process Communication
JCS	Joint Chief Staff
KPN	Kernal Process Node
LAM	Logging and Auditing Module
LAN	Local Area Network
LRM	Ada Language Reference Manual
MIP	Millions of Instructions Per Second
OS	WIS Operator System
PEM	Program Execution Module
PROM	Programmable Read-Only Memory
ROM	Read Only Memory
RPC	Remote Procedure Call
SSMM	Secondary Storage Management Module
TCB	Trusted Computing Base
TCP	Transmission Control Protocol
TM	Transaction Manager
WIS	WWMCCS Information System
WWMCCS	World Wide Military Command and Control

1.4 REFERENCES

[1815A 83] U.S. Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February, 1983.

[BERN 81] Bernstein, P. A. and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Survey* 13,2 (June 1981): 185-221.

[BIRR 83] Birrell, A.D. and B.J. Nelson, Communication Techniques for Remote Procedure Calls. In *Proceedings of the Ninth Symposium on Operating Systems Principles* (October, 1983).

[BIRR 84] Birrell, A. and B. Nelson, Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* 2, 1 (February, 1984).

[CAIS 85] U.S. Department of Defense, Military Standard Common APSE Interface Set (CAIS), Proposed MIL-STD-CAIS, 1985.

[CHAN 82] Chandy, K.M. and J. Misra, A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (August, 1982).

[CHER 84] Cheriton, D, The V Kernel: A Software Base for Distributed Systems. *IEEE Software* 1, 2 (April, 1984).

[CHOU 85] Choudhary, A., W. Kohler, J. Stankovic, and D. Towsley, A Distributed Deadlock Detection and Resolution Algorithm Based on Priorities, UMASS Technical Report, University of Massachusettes, Amherst, MA, 1985.

[ELMA 85] Elmagarmid, A.K. and M.T. Liu, Fault-Tolerant Deadlock Detection in Distributed Database Systems. In *Digest of Papers, 15th Annual International Symposium on Fault-Tolerant Computing* (June, 1985): 240-245.

[FRID 81] Fridrich, M. and W. Older, The FELIX File Server. In *Proceedings of the ACM 8th Symposium on Operating System Principles* (December 1981): 37-44.

[GRAY 79] Gray, J.N, Notes on Database Operating Systems. Operating Systems: An Advanced Course, Springer-Verlag, New York, (1979): 393-481.

[HAER 83] Haerder, T. and A. Reuter, Principles of Transaction-Oriented Database Recovery, *ACM Computing Survey* 15, 4 (December, 1983): 287-317.

[HO 82] Ho, Gary, and C.V. Ramamoorthy, Protocols for Deadlock Detection in Distributed Database Systems, *IEEE Transactions on Software Engineering* 8, 6 (November, 1982):.

[JACK 84] Jackson, B. and J. Salasin, Preliminary Requirements for the Army WWMCCS Information System (AWIS), Technical Report WP-84W00035, Mitre Corporation, 1984.

[JAGA 82] Jagannathan, J.R., and R. Vasudevan, A Distributed Deadlock Detection and Resolution Scheme Performance Study. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (1982).

[LAMP 78] Lamport, Leslie, Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21, 7 (July 1978): 558-565.

[NCSC 83] National Computer Security Center, Trusted Computer System Evaluation Criteria, CSC-STD-001-83, 1983.

[NELS 81] Nelson, B.J, Remote Procedure Call. PhD thesis, Carnegie-Mellon University, 1981. Published as CMU Technical Report CMU-CS-81-119.

[OBER 82] Obermarck, R, A Distributed Deadlock Detection Algorithm, *ACM Transactions on Databases* 7, 2 (June, 1982).

[RASH 81] Rashid, R., and G. Robertson, Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth Symposium on Operating System Principles* (Pacific Grove, CA, December 1981).

[SCHA 85] Schantz, R., R. Thomas, and G. Bono. The Architecture of the Cronus Distributed Operating System, BBN Laboratories, 1985. Submitted for publication, 1986.

[SINH 85] Sinha, M.K., and N. Natarajan, A Priority Based Distributed Deadlock Detection Algorithm, *IEEE Transactions on Software Engineering* SE-11, 1 (January, 1985).

[STAN 84a] Stankovic, J, Perspectives on Distributed Computer Systems, *IEEE Transactions on Computers* C-33 (December, 1984): 1102-1115.

[STAN 84b] Stankovic, J. and I. Sidhu, A Bidding Algorithm For Independent Processes, Clusters of Processes and Distributed Groups. In *Proceedings of the 4th International Conference on Distributed Computing Systems* (May, 1984).

[STAN 85] Stankovic, J, Stability and Distributed Scheduling Algorithms, *IEEE Transactions on Software Engineering* , (October, 1985).

[STUR 80] Sturgis, H.E., J. G. Mitchell, and J. Israel, Issues in Design and Use of a Distributed File System, *ACM Operating Systems Review* 14, 3(1980): 55-69.

[SVOB 84] Svobodova, L, File Servers for Network-Based Distributed Systems. *ACM Computing Survey* 16, 4(1984).

[THOM 79] Thomas, R.W, A Majority Concensus Approach to Concurrency Control for Multiple Copy Databases, *ACM Transactions on Database Systems* (June, 1970): 180-209.

[WALK 83] Walker, B.G., G. Popek, R. English, C. Kline, and G. Theil, The LOCUS Distributed Operating System. In *Proceedings of the ACM 9th Symposium on Operating System Principles* (October, 1983): 49-70.

[WELL 84] Wellings, A. J., D. Keeffe, and G. M. Tomlinson, A Problem with Ada and Resource Allocation. *Ada Letters* 3, 4 (January-February 1984).

2.0 OVERVIEW OF WIS OPERATING SYSTEM MODEL

The WIS OS is a distributed operating system in the sense that it provides a abstraction of a single system across network-connected multiple machines. When a host is running the full WIS OS software, it provides the same program interface. Hosts running other software are extended to provide the same generic network interface, i.e., they speak the standard WIS OS protocols.

Conceptually, the system is structured as three operating system levels:

a. Virtual memory: The WIS OS provides virtual memory spaces for program execution in (basically) the Von Neumann model. WIS OS determines the binding of virtual addresses to data, which should ideally provide the late binding and flexibility of so-called "mapped I/O".

b. Objects: WIS OS objects include open files, atomic transactions, jobs, processes and virtual spaces. Of course, language processors are free to define other types of objects either using these basic WIS OS objects or independently.

c. User entity: The WIS OS provides support for users to "encode" useful information and semantics in the system. In the simplest case, this is a fancy way of describing symbolic names for files; the semantics of the symbolic name for the file and the file contents are chosen by the user to correspond. In the more general case, the user should be able to associate a symbolic name with user-specified behavior according the desired semantics. One implementation of this facility would be to use symbolically named "trigger daemons", i.e., procedures invoked on reference to the symbolic name.

The primary focus of the WIS OS is to implement the object level. The virtual memory level is subsumed to some degree by operations on virtual spaces as objects. Internal management of virtual spaces is viewed primarily as the job of the language processors and individual programs. For example, the Ada compiler provides management of an address space and implementation of objects local to one address space. The user entity level is subsumed by a flexible naming and binding mechanism that allows flexible binding of names to semantics. Treating the semantic actions as objects (either processes or procedure invocations) and their bindings as objects allows the WIS OS to support this level but leave its actual implementation to higher-level software.

Each type of object is viewed as an abstract data type (ADT). For example, an open file viewed as a set of operations defined on it, similar to a stack (or other canonical examples of ADT). Objects are organized into a hierarchy of abstract data types, primarily to provide uniform interfaces to generic classes of objects and to define generic semantics at different levels. For example, an open disk file, terminal line and a UNIX-like pipe are all special cases of the generic open file. Thus, the generic operation READ should work with the same generic semantics on each. However, SEEK,PIPE_WRITER_QUERY and SET_ECHO might be operations specific to each of these three cases. Uniform interfaces of this ilk are preferred for the same reasons as device-independent I/O is attractive, i.e., late and flexibility binding plus simpler programming model.

For simplicity, efficiency and security, the WIS OS object implementation is structured into three levels:

a. Client stubs that provide a program-level abstraction that may extend the basic interface to the object. For example, special query operations may be implemented as run-time routines that invoke a more general query routine provided by WIS OS.

b. The kernel provides a fundamental object called a process plus operations on processes (including IPC communication). There may be other objects provided by the kernel if necessary.

c. Objects other than processes are implemented at the process level to minimize the size of the kernel. The kernel provides operations for invoking operations on objects between processes, such as a message-passing scheme.

The overall system structure includes these levels plus two others, namely application programs plus "agents", such as the user interface agent and electronic mail agent.

This layered model is an attempt to extract the consensus of how to build a system from a variety of distributed systems projects, including V (Stanford), Cronus (BBN), Accent (CMU) and Cedar/Mesa (Xerox PARC). In this model, we have used the term process loosely in the sense that an extra level of indirection (such as ports) might be introduced without significant change to the model. There are some issues to resolve as to the exact nature of a process as well.

3.0 WIS OS KERNAL

Section 3.0 presents a basic design for the WIS OS kernel. Significant departures from the basic design must have proper justification.

The WIS OS kernel is a distributed kernel that implements an abstraction of memory, processing and communication suitable for implementing CAIS-like process nodes as well as external file and structural nodes. The kernel is designed to provide minimal facilities required to meet security, fault-tolerance, performance and CAIS compatibility requirements. Facilities that need not be implemented in the kernel are implemented in server nodes that execute outside the kernel as well as by so-called run-time procedures that execute in the address space of the invoker. For instance, the CAIS function STANDARD_ERROR may be just a simple routine linked directly with the invoker, returning a value stored in the address space of the invoker. Other functions, such as SPAWN_PROCESS, use kernel facilities to create a process node (address space) and allocate the needed resources with the required protections and security control.

Figure 1 illustrates the layering of CAIS routines on the kernel implementation with a service module.

```
|  Ada Application Program  |  File Server Program  |
|---------------------------------------------------|
|  CAIS run-time routines   |                       |
|---------------------------------------------------|
|                   WIS OS Kernel                   |
|                                                   |
|---------------------------------------------------|
| machine 1 |   | machine 2  | . . .  | machine k
   |-----------------|-----------------|---------|
                  local network
```

Figure 1. Layering of CAIS

The kernel logically extends across all the machines in a WIS OS cluster (although in reality, a copy of the kernel (or some version) executes on each machine in the cluster.). The basic kernel interface is augmented by CAIS run-time routines to provide CAIS process node, structural node and file node operations. In the case of file nodes, there are one or more file server programs that actually implement files. The file server programs may execute on the same or different machines (or both) with transparent access to local and remote file nodes provided by the kernel facilities plus the CAIS run-time routines. In general, the kernel is viewed as an protected run-time library that implements routines and data structures that cannot, for security, fault-tolerance or performance reasons, be implemented in the CAIS run-time packages (that execute addressable to applications).

The kernel is intended for a hardware base of machines connected by a LAN, including mainframe hosts, workstations and server machines. The kernel is also designed to be reasonably portable over a large class of machines and local networks. The kernel in conjunction with CAIS support is intended to provide an efficient, real-time base on which to build sophisticated single-user systems, multi-user systems, network-accessed servers, and dedicated real-time applications. These applications may be distributed over one or more network nodes or workstations.

The kernel interface is characterized by describing the basic objects it implements plus the operations on these objects. These objects and operations are introduced in the following section.

3.1 Basic Kernel Model

The kernel is designed to support the CAIS standard node model, including discretionary and mandatory access control at B3 level. In addition, it supports the CAIS model (with some extensions) over a distributed hardware including multiprocessors as well as uniprocessors. Finally, it provides some basic support for fault-tolerance. The kernel is also designed to be portable across different machine architectures and networks. There are three basic classes of facilities provided by the kernel: memory, processing, and communication.

3.1.1 Kernel Process Nodes

A kernel process node (KPN) is a basic form of process node (as in CAIS). A KPN is a single address space with memory cells number 0 through N for some N. (We assume 32-bit addresses for now.) The use of an address in one KPN is independent in general of the use of the same address in another KPN. That is, two KPN's can be prevented from sharing any data directly by memory access. In fact, for security, a KPN at one security level may be unable to detect the existence of a KPN at another level. Operations are provided for changing the amount of valid memory associated with the KPN plus mapping portions of file nodes in and out of the address space, namely mapped I/O.

The WIS OS kernel implements KNP's or address spaces so that it can fully isolate different Ada programs for security and reliability reasons. Separate address spaces are used for numerous reasons:

a. Many machine architectures provide separate address spaces and this facility is required for proper use of the machines.

b. Separate address spaces arise between programs running on separate machines.

c. Separate address spaces provide low-level protection and security boundaries that can be enforced by a small amount of software compared to the compiler, for instance.

d. Separate address spaces simplify resource reclamation because all the memory in an address space can be reclaimed when the program terminates. This obviates the need for sophisticated, expensive and error-prone language-level automatic memory reclamation, as done (for example) in Cedar, the Xerox PARC programming environment.

A KPN contains one or more kernel tasks (ktask) that represent threads of execution control. These ktasks are used to implement Ada tasks. That is, each Ada task in a standard Ada program corresponds to a ktask unless it has been optimized into a procedure call from its invoker (as some implementations attempt to do). Ktasks are distinguished from Ada tasks because of differences arising between the kernel interface and the standard Ada interface. All applications are assumed (and required) to use the Ada interface which is then mapped to the kernel interface by run-time trap routines. Each ktask is identified by a cluster-wide unique ktask-id. This identifier is used by the Ada run-time to identify ktasks corresponding to Ada tasks. (A suggested size for this identifier is 32 bits.) Note: Ada tasks require kernel support to provide for multiprocessor scheduling of tasks within one Ada program.

The kernel also supports groups of ktasks, with operations from allocating a ktask-group-id, adding a ktask to a ktask-group, deleting a ktask from a ktask-group plus various query operations. Identifiers for ktask groups are a subset of the ktask-id's. Recommended is reserving a bit in the ktask-id space that indicates whether a single ktask or ktask group is identified.

For economy of name spaces, a KPN does not have its own identifier. Instead, it is identified in an operation by a ktask-id of a ktask in the KPN. The designation of the ktask versus the KPN is clear from the definition of the operation.

The kernel provides support for a per-ktask area by associating a ktask location and value with each ktask. Whenever a ktask is activated, the kernel stores its per-ktask value in its per-ktask location. As an expected use of this facility, each per-ktask value is a pointer to a standard per-ktask data area within the task's stack space. This is provided as an aid to Ada implementations of task, providing a pointer to a task-specific data that is set relevant to the currently executing task and processor.

Ktasks can be dynamically created and destroyed. It is assigned its unique ktask-id on creation that is used subsequently to specify that ktask. Also, it is created as part of the same KPN as its creator. A ktask is created in the initial state of awaiting-reply from its creating task. (See next section on intertask communication.) This provides the creator with the required control over the ktask to initialize its execution environment, including stack, parameters, etc., before the ktask begins execution. An Ada implementation may also create several such ktasks early in program execution to handle task references but only initiate execution when the task is "created" according to the execution of Ada program in question.

Intertask communication is provided in two forms by the kernel. First, tasks may send, receive, reply to, and forward messages. A ktask sends a message to another ktask-id (which may be a ktask group) and suspends execution until the message is received by at least one ktask and the ktask has replied to the message. If the specified ktask does not exist, the message transaction terminates with an error indication (which the Ada run-time can map into an exception). If the ktask-id represents a group, the sending ktask can receive additional reply messages using a GET_REPLY kernel operation. A task sending a message is said to be awaiting reply from the time it sends the message until the message is received and replied to by the receiving task, or else, an error is returned.

Second, a task can pass access to a single segment of memory in its KPN space to the recipient of its message. The recipient task can access this segment using intertask copy operations. The sender can specify read and/or write access to the sender. The presence of a segment in a message and its associated location, size and access is specified by reserved fields in the message.

The term message is used to indicate a basically untyped data array that is transferred from sender to receiver. Similarly, message transaction is an exchange of request and reply messages with zero or more intervening intertask copy operations to access segment data. The Ada run-time may implement rendezvous in terms of message transactions. Similarly, remote Ada procedure calls are implemented by sending the procedure parameters in a message to a well-known task for that procedure, which creates a ktask to execute the remotely executed procedure, allocating stack space, etc., and forwards the request message serving as a procedure invocation to the newly created ktask. On return, the ktask replies to the sending ktask and terminates or else takes on another procedure invocation.

Implementation note: there are performance advantages to using a fixed size, short message of a length such that most procedure calls can fit in their parameters. For example, 8 words or 32 bytes is reasonable. This allows the kernel to use a fixed size, statically allocated message buffer for each task. Data larger than a message is passed as part of the segment access mechanism using the intertask copy operations.

Support for naming in the kernel is limited to ktask groups. There are statically allocated ktask group identifiers that may be used as logical addressing of certain classes of services or facilities. A ktask requiring service in one such class can send to the associated ktask group and locate a ktask willing and able to provide the required service. In the simplest case, there is a ktask group corresponding to the remote procedure call binder service.

3.1.2 Miscellaneous

The kernel provides operations for reading the time, setting the time, delaying for a time period, and unblocking a delaying task. Time is maintained in SYSTEM.CLICKS units, which should be as fine-grain and accurate as practical.

For uniformity of communication (including device communication), devices are accessed through the device pseudo-task implemented by the kernel. Device operations are performed by sending messages to the device server task, which again appear at standard Ada procedure calls to application programs due to the interposition of run-time routines that map between the application interface and the kernel interface. The protocol used in these messages is the same as used for all other I/O activities in the system.

After the kernel has completed its internal initialization, it creates an initial KNP and an initial task on this KNP. The kernel may have to make some assumptions about the format of an Ada program module to start the first such KNP and its program.

3.1.3 Distributed Operation

All kernel operations that use ktask-ids work uniformly independent of machine boundaries. In particular, tasks on different machines may send and receive messages and access segments as though all tasks were executing on the same machine. Each KPN resides on exactly one machine at a time although the system may allow KPNs to migrate from one machine to another. Restricting a KPN to one machine avoids the problem of implementing shared.

3.1.4 Security

The kernel interface needs to provide access to and enforce the same multi-level security model provided in the protocol. That is, it should be possible to enforce the same level of security isolation between programs running in separate virtual spaces as between programs running on separate machines. The following model is proposed.

Supporting the CAIS model, each process node is created using a specified security level. Process nodes are divided into two classes, trusted and untrusted (correspondingly dividing ktasks into trusted and untrusted ktasks.) All ktasks in a KPN are trusted or untrusted depending on the status of the KPN. An untrusted process node cannot detect to any significant degree the existence or state of an untrusted process node operating at a different security level. (This is solving the confinement problem.) In particular, any attempt to send to a ktask at a different security level is equivalent in behavior to sending to a non-existent ktask. In contrast, trusted process nodes are allowed to pass information between security levels, including creating and managing process nodes at different security levels.

Trusted process nodes must run trusted software, as required for B3. Trusted process nodes may only run on certain trusted hosts. Unsecured personal workstations run at a single security level and only "see" other security levels by access negotiated indirectly through trusted processes. Other hosts may be authorized for running different security levels simultaneously but without having trusted processes. In this vein, file servers are envisioned for each level of security as well as multiple security level file servers. However, a workstation attempting access to another security level must go through a trusted process to "write down" the required data and cannot deal directly with a multi-level file server.

There needs to be some study of the cost of dealing with the confinement problem. Clearly, though eliminating the implicit communication channels altogether is infeasible, it is possible to interfere with such channels to minimize the effective data rate.

3.1.5 Fault Tolerance

There are three aspects to kernel support for fault tolerance. First, the kernel allows applications and server-level modules to provide mechanisms for detecting faults and taking remedial action. In particular, the application level can supply an exception handler that gains control of a task at the point it incurs a kernel-detected exception. The application is then free to terminate the task, attempt repair and restart the task or whatever is appropriate. As part of this support, the kernel ensures that no kernel operation has any effect on an application if an exception occurs in the midst of executing an "output" type of operation, such as sending a message. In this case, the message is not sent if there is an exception as part of sending the message. Similarly, the kernel ensures that a "read" type of operation, such as receiving a message, is repeatable if an exception occurs during the execution of the operation. For instance, one is able to re-receive a message that was being received when an exception was incurred. Thus, the intention is that the application will restart the operation after correction. These two statements are general principle may not be fully true. For instance, re-receiving a message from a task that has since terminated may be difficult to implement and have the wrong semantics.

A second area of fault tolerance for the kernel is dealing with hardware faults that affect its operation. Hardware failures include processor, memory, bus, network interface and other devices. On hard failure of any one of these components, there must be replacement or alternative component available to make continuation feasible. The kernel should be designed to use its operation from a point prior to the failure and continue with the alternative. In the case of transient failures, a similar mechanism should work to simply retry the operation with the same component. In many cases, a permanent component failure may only be detected by several failing retries under the assumption of the problem being transient failure.

Making the kernel operations atomic from a failure standpoint aids in kernel recovery from errors. The kernel can abort an operation in progress and simply retry the operation again from the beginning.

The final area to consider is software faults. Three solutions are proposed to handling these faults:

a. Fault-handlers: Retry operations in the case where the problem may be a software timing error and not a hard coding error.

b. Data structure audit: Audit data structures for problems, both by periodic checks and as part of operations. Diagnosis and repair is attempted with some possibility of loss of a kernel operation or messages, if necessary.

c. Replicated code: Refresh code from Programmable Read-Only Memory (PROM) or disk when corruption appears possible. Alternatively, the code could be protected from corruption by memory protection or by being burned in Read Only Memory (ROM).

The kernel fault handling must be analyzed for behavior under different faults to establish the properties of the fault handlers with respect to correct operation. There is a danger that fault handlers will simply delay recovery from a fault, i.e., a fast reboot is better than a slow one. Second, there is a danger that a fault handler may not repair the damage correctly and lead to the system continuing in an apparently error-free fashion which is in fact more detrimental to overall system behavior than total kernel failure at the point of the original fault.

3.2 Kernel Operations

The following description of operations is intended to be illustrative and is taken fairly directly from the V kernel, which has heavily influenced this design. For simplicity, we use simply "task" for ktask and assume it is understood that we are dealing at the kernel interface level, not the Ada program level.

The operations provided by the kernel can be divided into three classes: kernel traps, kerneltask operations, and kernel device operations. The most basic kernel operations, including SEND, are implemented as kernel traps. These operations are invoked by executing a trap or system call instruction which invokes the kernel. A number of secondary operations are implemented by a pseudo-task running in the kernel, called the kerneltask. Such operations are invoked by sending to the kerneltask's task-id. Finally, operations on kernel-implemented devices are provided by a second pseudo-task, called the kernel device server. Such operations are invoked by sending messages to the device server's task-id, using the standard I/O protocol.

The kernel traps include:

COPY_FROM	COPY_TO	FORWARD
FORWARDER	GET_REPLY	QUERY_KERNEL
RECEIVE_ANY	RECEIVE-SPECIFIC	REPLY
SEND	TASK_DELAY	

The kernel task operations include:

ADD_TO_GROUP	BIND_REGION
CREATE_HOST	CREATE_JOB
CREATE_PROCESS_NODE	CREATE_TASK
DESTROY_TASK	GET_PAGE_SIZE
GET_TIME	INTERRUPT_TASK
QUERY_PROCESS_STATE	QUERY_TASK_GROUP
QUERY_TASK_STATE	REMOVE_FROM_GROUP
SET_EXCEPTION_HANDLER	SET_PROCESS_NODE_PRIORITY
SET_TASK_PRIORITY	SET_TIME
TASK_CREATOR	UNBIND_REGION
VALID_TASK_ID	WAKEUP
WRITE_TASK_STATE	

In the following descriptions, the active task or invoking task always refers to the task that executed the kernel operation.

Some operations such as SET_PROCESS_NODE_PRIORITY and SET_TIME are intended to be used only by "operating system" or management tasks and should not be used by application programs. Again, these kernel operations are to be viewed as basic "extended machine" instructions to be used by the Ada run-time environment to implement the standard Ada tasking model plus provide remote Ada procedure calls to access remote facilities.

3.2.1 Task and Process Management

CREATE_PROCESS (parameters): Creates a new task with the specified parameters and return its unique task identifier. Parameters should be chosen to match those required by CAIS. Parameters may include priority, initial program counter and initial stack pointer. Note: task must execute at same security level as creator since it is contained in some process node.

The task is created awaiting reply from the invoking task an in the same process node. The segment access is set up to provide read and write access to the entire process space of the newly created task. The creator must reply to the newly created task before it can execute.

CREATE_PROCESS_NODE (parameters): Creates a new process node with the specified parameters as required for CAIS support including access control, security level and possibility attributes. Other parameters to SPAWN_PROCESS can be passed as parameters into the address space of the KPN since there is no protection issue involved with these parameters. A parameter specified whether this is a new job or not. There is not a separate kernel primitive for creating new jobs. (It is not clear that the kernel even needs to make this distinction.) A first or root task is created on the new process node and that task-id of the task is returned.

CREATE_PROCESS_NODE is similar to CREATE_TASK except the new task is created on a new KPN. The new KPN initially has a null address space. It is intended that the creator of the team initializes the memory address space and root task state using BIND_REGION, MOVE_TO, and WRITE-TASK_STATE.

DESTROY_TASK (TASKID): Destroys the specified task and all tasks that it created. When a task is destroyed, it stops executing, its task-id becomes invalid, and all tasks blocked on it become unblocked (eventually). When all the tasks in a process node are destroyed, the process node is automatically deleted.

QUERY_PROCESS_STATE (TASKID, ACCESS_TO_PROCESS_STATE): Copies the state of the process node associated with the specified task into the structure that can be accessed (i.e., pointed to) by ACCESS_TO_PROCESS_STATE. The state includes whether fields are required as part of the process implementation and include START_TIME, FINISH_TIME, MACHINE_TIME, etc., as required by the CAIS specification.

QUERY_TASK_STATE (TASKID, ACCESS_TO_TASK_STATE): Copies the state of the specified task into the structure that can be accessed (i.e., pointed to) by ACCESS_TO_TASK_STATE. This includes registers, state, etc.. The message buffer is only available if TASKID is the invoking task or is awaiting reply from the invoking task. If not, the appropriate fields in the structure are zeroed.

SET_PROCESS_NODE_PRIORITY (TASKID, PRIORITY): Sets the priority of the process node associated with TASKID to the specified priority and return the previous priority. Each task effectively runs with the absolute scheduling priority of its process node plus the priority specified when the task was created. SET_PROCESS_PRIORITY changes he absolute scheduling priority of each task on the process by modifying the process priority. This operation is intendefor implementing macro-level scheduling and may eventually be restricted in use to trusted or privileged process nodes. Note: changing priority is intended as the means for implementing SUSPEND_PROCESS and RESUME_PROCESS in CAIS. There should be at least one low level of priority at which a process is guaranteed not to execute.

SET_TASK_PRIORITY (TASKID, PRIORITY): Sets the task's execution priority.

TASK_CREATOR (TASKID): Returns the task id of the task that created TASKID.

VALID_TASKID (TASKID): Returns TRUE if TASKID is a valid task identifier; otherwise returns FALSE.

WRITE_TASK_STATE (TASKID, STATE): Copies the specified task state record into the kernel state of the task specified by TASKID. The specified task must be the invoking task or awaiting reply from the invoking task. The kernel checks that the new state cannot compromise the integrity or security of the kernel.

3.2.2 Exception Handling

The kernel provides minimal exception handling. When a task incurs an exception, it is logically forwarded to the exception-handler task associated with this task. At that point, the exception-handler task has full control of the task and may diagnosis the problem, take remedial action, terminate the task, or continue the task. There are two basic operations in the kernel, SET_EXCEPTION_HANDLER and INTERRUPT_TASK:

INTERRUPT_TASK (TASKID, parameters): Causes the specified task to suspend execution by sending a message to the invoking task with the specified parameters.

SET_EXCEPTION_HANDLER (TASKID, EXCEPT_HANDLER): Sets the exception handler for the specified task to be the task specified by EXCEPT_HANDLER. A task inherits the exception handler of its creator by default.

3.2.3 Memory Management

BIND_REGION (TASKID, ADDR, SIZE, FILE_HANDLE, OFFSET, MODE): Binds the addresses in the region from ADDR to ADDR+SIZE to the file node specified by FILE_HANDLE starting at OFFSET. Any previously bound pages are unbound.

GET_PAGE_SIZE: Returns the number of bytes in a page.

UNBIND REGION (TASKID, ADDR, SIZE): Removes the bindings established by BIND_REGION for the pages in the specified range. (References in the address range now generate address exceptions.)

3.2.4 Task Groups

ADD_TO_GROUP (TASK_GROUPID, TASKID): Adds the task to the specified task group.

QUERY_TASK_GROUP (TASK_GROUPID, TASK_INFO): Returns information about the specified task group, including number of members, etc.

REMOVE_FROM_GROUP (TASK_GROUPID, TASKID): Removes the task from the specified task group.

3.2.5 Inter-Task Communication

COPY_FROM (SRC_TASKID, DEST, SRC, COUNT): Copies COUNT bytes from the memory segment starting at SRC in the process space of SRC_TASKID to the segment starting at DEST in the invoking task's space. The SRC_TASKID task must be awaiting reply from the invoking task and must have provided read access to the segment of memory in its space using the message format convention described for SEND.

COPY_TO (DEST_TASKID, DEST, SRC, COUNT): Copies COUNT bytes from the segment starting at SRC in the invoking task's process node space to the segment starting at DEST in the process node space of the DEST_TASKID task, and return the standard system reply code OK. The DEST_TASKID task must be awaiting reply from the invoking ask and must have provided write access to the segment of memory in its space using the message format conventions described under SEND.

FORWARD (MSG, FROM_TASKID, TO_TASKID): Forwards the message pointed to by MSG to the task specified by TO_TASKID as though it had been sent by the task FROM_TASKID. The task specified by FROM_TASKID must be awaiting reply from the invoking task. The effect of this operation is the same as if FROM_TASKID had sent directly to TO_TASKID, except that the invoking task is noted as the forwarder of the message. Note that FORWARD does not block.

FORWARDER (TASKID): Returns the task id that forwarded the last message received from TASKID, providing TASKID is still awaiting reply from the invoking task. If the message was not forwarded, TASKID is returned. If TASKID does not exist or is not awaiting reply from the invoking task, 0 is returned.

GET_REPLY (MSG, TIMEOUT): Returns the next reply message returned in response to the last SEND operation assuming one is queued or is returned within the next timeout clicks. GET_REPLY is only of use when the message was sent to a task group, thus providing for multiple reply messages.

RECEIVE_ANY (MSG, SEGMENT, SEGMENT_SIZE): Suspends the invoking task until a message is available from a sending task, returning the task-id of this task, and placing the message in the array (pointed to by) MSG. At most the first SEGMENT_SIZE bytes of the segment included with the message is placed in the buffer starting at SEGMENT. The actual number of bytes in the portion of the segment received is returned in SEGMENT_SIZE.

RECEIVE_SPECIFIC (MSG, TASKID, SEGMENT, SEGMENT_SIZE): The same as for RECEIVE_ANY except that a message is only accepted from the specified task. If the specified task does not exist or is subsequently destroyed, RECEIVE_SPECIFIC returns an error status.

REPLY (MSG, TASKID, SRC, DEST, BYTES): Returns the specified reply message and segment to the task specified by TASKID. The specified task must be awaiting reply from the invoking task. the specified reply message and segment to the task specified by TASKID and return TASKID.

SEND (MSG, REPLY_MSG, TASKID, SEGMENT, SEGSIZE, ACCESS, TIMEOUT): Sends the message in MSG to the specified task providing it with the specified ACCESS to the SEGMENT and blocking the invoking task until the message is both received and replied to. If SEND completes the process successfully, it returns the task-id of the task that replied to the message. The task-id returned differs from that specified in the call if the message is forwarded by the receiver to another task that in turn replies to it. If the send fails (for instance, because the intended receiver does not exist), SEND returns the task-id of the task the message was last forwarded to (the task-id it was sent to, if it was never forwarded). Access to segment is read/or write. The TIMEOUT parameter specifies the amount of time the sender will wait before the message is received without canceling the message transaction. This is to support conditional and timed entry calls.

Several "optimizations" are possible. In the V kernel, the REPLY_MSG and MSG pointers were made the same for economy. Similarly, the segment specification was encoded in the message since this information was generally of interest to the recipient as well as the kernel (which has to enforce the segment access). Finally, an initial portion of the segment is transmitted to the recipient when read access is provided, as suggested when RECEIVE_ANY or RECEIVE_SPECIFIC specify an segment buffer.

3.2.6 Time Functions

GET_TIME (TIME, CLICKS): Returns the current time in seconds and clicks.

SET_TIME (SECONDS, CLICKS): Sets the kernel-maintained time to that specified by SECONDS and CLICKS.

TASK_DELAY (SECONDS, CLICKS): Suspends the execution of the invoking task for the specified number of seconds and clicks where a click is one microsecond. (The accuracy of TASK_DELAY reflects the machine clock resolution.) The task may be unblocked by WAKEUP.

WAKEUP (TASKID): Unblocks the specified task if it is delaying using TASK_DELAY.

3.2.7 Miscellaneous

QUERY_KERNEL (TASKID, GROUP_SELECT, REPLY): Queries the kernel on the host where task TASKID is resident. The GROUP_SELECT field specifies what information is to be returned in the REPLY message. The available group selection codes are MACHINE_CONFIG, to return information about the processor configuration, PERIPHERAL_CONFIG, to return a list of peripherals available on the machine, KERNEL_CONFIG, to return the kernel's configuration parameters, MEMORY_STATS, to return memory usage statistics, and KERNEL_STATS, to return other kernel statistics. These codes, and the corresponding structures that may be returned, are defined in the standard interface file.

Again, these operations are intended to be suggestive of the types of kernel operations expected. The actual kernel operations, their parameters and semantics, should be modified to support the CAIS environment. Moreover, none of these kernel operations should be directly visible to Ada programmers in general. They instead deal with standard CAIS primitives (although various extensions, such as for dealing with task groups may be required).

3.2.8 Device Management

Basic device support is provided by the kernel. For instance, the kernel should provide basic raw access to disk, network interfaces, tape, console, etc. All I/O operations are to be performed across address space boundaries using the SEND operation to communicate in the remote procedure call paradigm. Ada I/O operations are implemented as stub routines that generate messages to tasks on process nodes implementing various I/O services. Keeping with this model, the kernel provides a set of device pseudo tasks that implement the devices. At the kernel interface, device I/O is performed by sending messages to these device tasks, the same as I/O to the file servers, printers and other I/O services. That is, the same message protocol should be used for all Ada I/O procedures.

3.3 Kernel Implementation

The WIS OS kernel should be a distributed kernel in that a single-system image should be provided to programs across all machines in one interconnected domain running the WIS OS kernel. Machines running a guest-level implementation of the WIS OS protocols may appear externally to be part of this machine-transparent domain, but do not generally appear that way to programs running in the machine.

The objective of this project is to design the WIS OS kernel interface and implement a prototype kernel that provides this interface as well as implements the WIS OS kernel protocol. This prototype should be developed to provide an indication of the performance of the protocol and the cost of implementing the interface. It would also provide experience with structuring the kernel for portability.

The prototype kernel could be developed for a medium-performance workstation, such as the SUN, Apollo or Perq. Based on sizes of similar kernels such as the V kernel, Accent kernel, etc., we would expect the following:

a. The kernel would be about 20,000 lines of source code.

b. The kernel could be developed almost entirely in the Ada language if a suitable compiler is available.

c. The kernel would take about two man-years to produce.

Recommended is that an extra three man-years to be allocated to experimenting with the performance, reliability and security aspects of the kernel. Some of this effort might be used to support use of the kernel and respond to problems found in the course of performing the other tasks as in the following sections. A suggested structure to the kernel is as follows. The performance-critical) inter-task communication operations should be implemented as highly optimized traps into the appropriate routines. (For instance, one operation per trap location for machines providing trap vectors.) The remaining operations for general process, task, memory and device management should be implemente by internal-to-kernel tasks that are invoked (across the kernel interface) by the SEND operation. This design means that implementing these operations on remote objects (on another machine running the kernel) is relatively simple. That is, assuming the SEND operation is implemented transparently between machines, a DESTROY_TASK operation (for example) is implemented by sending the message to the remote machine on which the specified task is executing.

Each task is represented by a small data record linked to its associated process node descriptor. The process node descriptor contains the address space information and other per-process information. A separate data structure is required for task group membership plus the usual data structures for virtual memory management and device management. However, the space cost per task should be very low, dominated by the task descriptor space required for processor state.

The kernel should execute on each machine participating in the distributed kernel abstraction. Kernel operations on remote processes are realized by communication between kernels using an inter-kernel protocol. Aspects of this protocol are described in the following sections.

3.4 Inter-Kernel Protocol

IPC is a key performance issue in systems of this design. With attention to detail, the cost of moving data (copying) becomes the dominant cost for all but small amounts of data. The intertask communication be efficient enough such that it can be used by diskless workstations for network file access and (demand) paging.

The WIS OS kernel protocol or protocols provides transport-level communication between tasks running the WIS OS kernel, or possibly "appearing" to run the WIS OS kernel. For instance, existing systems may be extended to "speak" this protocol, and thereby participate as part of WIS OS.

The kernel protocol must have the following important attributes:

a. Efficiency: Provide efficient transaction or request-response style interaction between machines, as is typical on a local network cluster of machines.

b. Reliability: Provide a range of reliability with extensive error reporting facility. The kernel protocol should report why communication failed rather than just that it did fail.

c. Security: Provide for secure (encrypted) communication of data both with Data Encryption Standard (DES) style encryption and public-key encryption systems.

d. Real-time: Provide for real-time communication, i.e., where getting the most recent information is more important than getting all information. As part of this, there should be a priority associated with communication as well.

Some of the requirements closely overlap that provided by Transmission Control Protocol/Internel Protocol (TCP)/(IP). There is also currently an Internet Task Force (End-to-End Services) that is proposing to develop a new "transaction-based" protocol for the Internet that would be even more appropriate, probably similar to RDP.

In general, the inter kernel protocol seems best as based directly on IP datagrams using a general request-response paradigm, as used in the V kernel and various remote procedure call protocols. Basically, a request consists of one or more datagram packets. A response is one or more datagram packets sent back to the sender of the request. The response is viewed as an acknowledge that the request was received as well as a reply to the request. Requests are retransmitted some number of times until acknowledged. Various "response-pending" packets are used as well. Finally, the intertask copy operations can be implemented as multiple packet requests and responses as well. The V kernel protocol as well as the various so-called "blast" protocols used at MIT serve as reasonable models for this aspect of the protocol.

Communication with a group of tasks should map to multicast datagrams that are logically addressed to all hosts involved. A proposal is being developed in DARPA for a multicast extension to IP, suitable for supporting this form of communication within the standard DoD Internet.

3.5 Concluding Remarks

The kernel is designed to provide a small, high-performance base on which to provide standard Ada tasking and CAIS process nodes. The kernel as a separate entity from the Ada runtime is required to support multiple address spaces, represented as process nodes. The small size of the kernel is intended to allow it to run on all machines, although possibly in different configurations. With the layering of Ada and CAIS run-time routines on top of the kernel, the higher levels of software have a network transparent base on which to build higher level services, such as file and database service, printer service, multi-window systems and user interfaces, not to mention WIS applications.

The Ada run-time layer is provided by the Ada development team. Some aspects of the mapping onto this kernel are described in Section 4.0. This includes protocols for use with the message primitives as well as how naming is done.

It appears not feasible to provide a small, efficient distributed kernel as proposed here that meets the security and fault-tolerance constraints of WIS and operates over a wide area network. WIS is assumed to be segmented into clusters that represent domains of relative trust, high data rate connectivity and common administration. The kernel provides relatively transparent operation within a cluster. Interaction between clusters is provided by cluster gateways that serve as translators, isolators and locators between the local cluster and the outside world. This is described further in Section 5.0.

4.0 ADA REMOTE PROCEDURE CALL

The WIS Ada remote procedure call facility provides the ability to invoke Ada procedures on a remote machine (more or less) as though they were being invoked locally. This supports a key goal of the WIS OS, namely allowing applications to be written in the Ada language and be relatively independent of the distributed hardware base on which the system runs. For example, a new database facility is defined in terms of its Ada procedural interface. Clients of this facility invoke these procedures, oblivious to whether the facility is running locally or remotely on a separate machine in the cluster, or possibly another cluster. Similarly, a module can be written with considerable independence of the clients or invokers of the module being local or remote.

There are four components to the Ada remote procedure call facility:

a. Presentation Protocol: Specifies how various types of parameters are to be represented in packets or messages.

b. Stub generator: Generates stub routines that map the call onto a packet or message at the client end plus map a packet onto a particular procedure, stack and activation record at the server (invokee) end.

c. Binder: Matches calls up with the call definitions.

d. Transport mechanism: Delivers the bits reliably between the call and the callee.

The transport level facility is provided by the kernel. That is, procedure arguments and return values are stored in messages and transported between the caller and the callee using the message-passing primitives. For further details on the kernel, the reader is referred to Section 3.0.

4.1 Presentation Protocol

There are three major design considerations in a presentation protocol:

a. Is there one network standard representation for data of a given type or several, depending on machine, etc.?

b. Is data explicitly typed in a message or is it implicit based on the "type" of the message? This might be regarded more as a granularity question.

c. How extensive is the protocol in covering different types? For example, does it handle arbitrary structured types, user-defined types, etc., or does it only define representations for basic types like integer, pointer, etc.?

Providing complete representation for all Ada types is very ambitious. Therefore, the goal is a modest protocol that handles the common types efficiently.

A presentation protocol should be developed that specifies a representation for the base data types in the Ada language, including integers, unsigned numbers, floating point numbers and access types (pointers). Required is that this protocol follow the Xerox Courier protocol to the closest degree possible. Deviations and extensions to Courier in the Ada presentation protocol should be clearly justified.

4.2 Stub Generator

The stub generator is a program that "compiles" Ada module interfaces and generates stub routines for both the client (invoker) and server (module) ends, using the kernel facilities to communicate between these ends when separated by a network or calling across virtual address boundaries. The stub generator takes as input an Ada module interface. The client stubs are a set of Ada routines generated by the stub generator that perform several functions.

First, the client take their parameters and pack them into a message according to the presentation protocol. The message includes a unique identifier for the procedure to be invoked, probably structured as (module, procedure). At minimum, a stub should handle all non-reference parameters plus reference parameters that refer to an object that itself does not contain references. Extensions to handle more complex data types are invited. However, the stub generator should recognize and report any procedural interfaces that it cannot handle correctly.

Second, the stub routine uses the binding facilities and the kernel facilities to locate the server machine and stub, and communicate with the server end. The kernel communication facilities suspend the task during the communication and then cause it to resume when the remote procedure call returns.

Finally, on return the stub routine unpackages the return values and places them in the appropriate locations for a normal procedural return. Of course, the stub must handle the propagation of Ada exception signals from the remote call back to the caller.

The stub generator should provide the same mapping to messages for the same procedure interface specifications so as to support uniform protocols. For example, the PRINTER.READ, FILE.READ, PIPE.READ and TERMINAL.READ operations should have the same parameters, as required for a uniform I/O interface.

The server stub performs a complementary set of facilities. First, as part of initialization it registers the availability of the module with thebinding facility. This allows the module to be located by remote calls.

Second, the server stub accepts incoming packets for the module, allocates the call to a process or task, translates the packet contents into the appropriate activation record, and gets the allocated task to invoke the appropriate procedure. The allocated task is set up as a representative of the remote invoking task; that is, its associated account and permissions appear as that of the remote invoking task.

Finally, on procedure return (or exception), the local task takes the return values, places them in a message, and sends it back to the invoker.

The server end could be structured as a manager process that receives all incoming Remote Procedure Call (RPC) packets. It employs a set of helper processes that actually execute the procedure calls. When a RPC message comes in, it allocates a helper process to execute it and passes the message and pointer to the server stub to the helper process.

The helper process then invokes the server stub which simply retracts the parameters from the message and invokes the procedure, plus handling the return as above. There may be a queue of helper processes for this purpose, reducing the overhead of creating a new process on each call. This structure may be replicated in each address space providing remote invocation so that the binding of helper processes to server stubs is straight forward, i.e. the server stubs are bound in with the module they serve.

4.3 Binding Facility

The binding facility connects procedure invocations with their remote server stubs. This binding can take place at a number of different times ranging from system initialization to program initialization to first invocation to each invocation. The binding can be performed with the aid of a central name service or by a decentralized facility, such as that provided by the kernel group communication facility.

A binding facility should be provided that gives efficient, robust binding. The following basic design is recommended: the stub generator allocates a unique identifier for each module and each external procedure within the module. Stub routines for the client and server stubs are generated knowing this unique identification. When a module that provides remote invocation is initialized, it registers the unique identifiers of its exported procedures with the local remote procedure call manager. Each client application/address space maintains a table of bindings from procedures to specific remote procedure call managers. A client stub uses this table to map to the appropriate remote procedure call manager when sending off the invocation message. If the entry for that particular procedure is invalid, the stub routine uses the kernel multicast facility to query all the remote procedure call managers to determine who implements this procedure, again using the unique identifiers. Assuming it gets a response, it stores the RPC manager identifier in the table and sends to the manager to invoke the desired procedure. This design corresponds to binding on first use except that rebinding should be supported on failure, i.e., the original RPC manager failed.

Also recommended is a simple facility be provided so that the programmer can initiate the binding at program initialization as well. Assuming a table of remote stubs is generated and bound into the program from the stub generator, this only requires a procedure that can be called, which systematically goes through the stub table and sets up bindings for each stub. This is more efficient and also provides earlier detection of binding failures. One can determine in advance that a given module is not available, rather than waiting until half way through program execution.

4.4 Transport Mechanism

TBD

4.5 Summary

The key goal is to provide a clean, efficient remote procedure call mechanism for the Ada program. The ideal of complete transparent procedure calls seems unattainable in the short term.

Of primary importance is the concentration on providing the subset of capabilities of maximum utility that can demonstrate the feasibility of this approach. In this vein, the RPC system should allow for programmer-generated stub routines to be mixed with the automatically generated stubs. This allows extensions by applications beyond the RPC facilities available at the time plus hand optimization of performance critical routines as appropriate.

The work of Nelson and Birrell [NELS 81] [BIRR 84] [BIRR 83] on remote procedure calls should be studied as should the Courier remote procedure call mechanism. Other relevant reports are appearing in the computer system research literature.

5.0 CLUSTER GATEWAYS

Each cluster has one or more gateway machines that connect the cluster to wide area networks, and therefore to other clusters. Unlike conventional datagram gateways that simply provide packet routing (and little else), cluster gateways are required to take an active part in insulating the cluster from outside considerations. These considerations include differences in communication protocols and characteristics, security and reliability assumptions. That is, the gateway really does have to function as a "gate" that can close selectively, not just a mindless pipe to the outside.

For efficiency within a cluster, network communication is optimized for local network characteristics or logical local networks. A logical local network is one or more physical local networks connected by bridges such that the existence of multiple physical network is (more or less) transparent to the hosts connected to the network. For communication outside the cluster, the gateway implements a local alias task that represents the remote task with which communication is to take place. Similarly, communication coming into a local network is handled as originating from an alias task in the gateway. This basic model has several benefits.

First, communication with tasks outside of the cluster appears the same as communication with a local task because of the local alias task. Thus, there is no need to compromise local communication to make wide area communication possible. Consequently, local cluster communication is very efficient.

Second, inter-cluster communication cannot occur without the gateways agreeing to create the requisite alias tasks. Because the creation of alias tasks can be handled at a fairly high level, this provides a reasonable security control over inter-cluster communication.

Finally, because the gateway is explicitly involved in translating between local communication and wide area communication, it can serve to isolate the local network from failures and errors, as well as contain local cluster failures.

A cluster gateway is structured as a half-gateway in the sense that it translates from the local network protocol to an internetwork protocol (in the truest sense of the term). That is, the internetwork protocol is primarily used between networks as opposed to covering a collection of networks. On the local network side, the gateway appears as a server that provides access to extra-cluster facilities. On the internetwork side, the gateway appears similarly with the addition of explicit data transfer primitives and a more general data packet interface. The next section describes the server interface.

5.1 The Gateway Server

The gateway server supports the following basic operations:

> CREATE_LOCAL_ALIAS (CLUSTER, REMOTE_TASKID): Creates a local alias task for the specified remote task and return a task-id for this local alias. Subsequently, a local task can communicate with this remote task by sending to the local alias.
>
> CREATE_REMOTE_ALIAS (LOCAL_TASKID, CLUSTER): Creates an alias for the local task on the specified remote cluster. Tasks on the remote cluster are now able to communicate with this local task.

ADD_TO_LOCAL_GROUP (TASK_GROUPID, LOCAL_ALIAS): Adds the local alias task to the specified task group. Messages sent to this group are subsequently forwarded to the remote task corresponding to this local alias.

REMOVE_FROM_LOCAL_GROUP (TASK_GROUPID, LOCAL_ALIAS): Removes the local alias task from the specified task group.

ADD_TO_REMOTE_GROUP (TASK_GROUPID, CLUSTER, REMOTE_ALIAS): Adds the remote alias to the taskgroup in the specified cluster. This is equivalent to ADD_TO_LOCAL_GROUP except the request is sent to a remote gateway and permission checking is more vigorous.

DEFINE_INTERCLUSTER_SERVICE (NAME, RANGE, SERVICE, TASKID): Define a new intercluster service with the specified name as part of the intercluster directory. This service is then accessible by name from any cluster within the specified range of clusters.

UNDEFINE_INTERCLUSTER_SERVICE (NAME): The specified service is removed from the gateway intercluster directory.

The basic operation envisioned is as follows. Various services within a cluster register themselves with a cluster gateway in the intercluster directory. This directory is logically one directory but in reality is a distributed database. The registration includes security level, scope of definition (which clusters may access this service), a description of the service, and the tasks or task groups that implement the service.

The name space of the gateway cluster directory is added to the name space of file and other objects of the local cluster. A client in the local cluster can query the intercluster directory for services that are available. It can then access such a service by creating a local alias for the task or task group representing this service and then proceed to use this service as though it is local. In the course of creating the local alias and setting up a remote alias for the local client task, the gateways would perform the required authentication on the client tasks involved.

In some cases, the service may need to pass client request servicing off onto other secondary tasks in the same process node (or a different process node). In this case, when dealing with a remote client task, the server uses CREATE_REMOTE_ALIAS to create a remote alias for this secondary task. The server then returns this secondary alias id and directs the client to use this for former communication. For example, a file server may have a single task for file name lookup and file open. However, subordinate tasks may handle all subsequent I/O requests.

In this fashion, clients can access remote services with no complication over accessing local services. Moreover, gateways are provided with a firm basis for implementing access control, namely their control of alias tasks.

5.2 Internetwork Protocol

The internetwork protocol used between cluster gateways or half gateways comprises several layers. First, there is a transport layer for delivery data between cluster gateways. The selection for transport layer depends highly on the interconnection between gateway halfs. For instance, two gateway halfs may be connected by a shared memory board as a communication channel. In this case, a simple transport protocol suffices. More commonly, the cluster gateways may communicate over long-distance links, possibly traversing other networks. In this case, a protocol such as TCP seems applicable.

The second level is a service protocol that allows multiple data streams between remote tasks to be multiplexed on a single intergateway connection. This optimizes the transfer of data from gateway to gateway.

Finally, one of the data streams is a control stream for one gateway to make requests of another. A control stream in the opposite direction allows the remote gateway to respond to these requests.

5.3 Implementation Details

The gateway server registers as a service in a cluster the same as other services, running on top of a standard kernel. The kernel must allow the gateway server to receive packets addressed to alias tasks as well as sent to any groups to which alias tasks belong. The kernel must, in addition, allow the gateway server to send out inter-kernel packets to local tasks.

Basically, the gateway server implements the inter-kernel protocol for local alias tasks, receiving packets addressed to them, taking the appropriate action, and then responding.

When a "Send" packet is received for an alias process, the gateway server checks whether this is a retransmission, and if not, transmits the packet on to the remote cluster and task associated with this local alias. The remote gateway, on receiving the "Send" packet, translates it to the appropriate local task, translates the sender task-id into a local-to-this-gateway task-id and forwards the packet on to the local task as though it was a local SEND. Retransmissions are filtered out if the connection between cluster gateways is reliable. In any case, the rate of retransmission across wide area links can be modified by the gateways to match the performance characteristics, independent of timeout values and retransmissions on the local network cluster.

The cluster gateways could make good use of an internet multicast facility, as is being currently proposed.

The contractor should develop a specific design based on this general outline with attention to security, fault-tolerance and general survivability. Ideally, the intercluster directory should make use of distributed integrated database technology developed as part of the rest of WIS.

6.0 OS SUPPORT FOR DATABASES

This design is a modification of a V I/O file access design mechanism adapted to DBMS requirements as identified in discussions between David Cheriton and Gio Wiederhold (both of Stanford University) in March of 1985.

6.1 Basic Model

The database management system (DBMS) executes as a set of tasks provided by the operating system. It defines an Ada procedural interface for clients. Thus, client programs invoke the DBMS either locally or remotely using the procedure call interface, with remote Ada procedure calls if the DBMS is not executing locally. The DBMS requires certain facilities of the operating system, including basic program execution facilities, reasonably accurate time and (most importantly) file support. It also requires an efficient, reliable file system supporting file locking, page-level locking, replication and recovery, and atomic update.

Each file is a sequence of fixed-size "blocks" or pages whose size is know at DBMS compile time. For efficiency reasons, the blocks should be 1 KB or larger with 4 KB looking very attractive. The blocks associated with a file represent the data most recently committed to that file.

To access a file, a file is "opened". The open file represents a "version" of the file blocks. There are two options here. The open file may represent the data blocks as they were (those committed) at the time the file was opened except for any modifications made using this open file. In particular, changes (even if committed) made via other open files are not apparent in this open file. The alternative mode is one in which changes to file pages appear in the open file once committed by others in addition to any changes made to this open file. These two modes are perceptually equivalent under appropriate locking assumptions. An open file is the unit of atomic update/abort provided by the file system, as part of the CLOSE operation. File closing also releases all locks. The file system also implements a SAVE_POINT operation that effectively provides a commit without losing access to the file, as with the CLOSE.

The following is a sketch of the operations with a few comments on semantics and issues. Note: there may be a need to reconcile these primitives with CAIS. These can at least be made upwardly compatible.

6.2 File Management Operations

The following operations are required for creating, deleting, querying and modifying files.

> DEFINE_FILE (NAME, REPLICATION, RECOVERY, SECURITY, ALLOCATION): Creates a file with the specified name and associated attributes. The replication parameters (to be further defined) specify the degree of replication and suggested replication sites. The recovery parameters specify the permanence of logging and recovery information for updates. For example, temporary files might have no logging. The security parameters specify the security classification of the file. The allocation parameters can request initial allocation unit, can ask that the file be contained within a single cylinder (or adjacent cylinders), and can request co-location of the file with another file. It appears that a best-effort support for allocation requests of co-location and cylinder containment suffice since this is simply a performance improvement for the DBMS.

QUERY_FILE (NAME, REQUESTED_INFO, RETURN_BUFFER): Returns the requested information about the specified file, including attributes of its replication, recovery, security, and allocation.

MODIFY_FILE (NAME, ATTRIBUTE, NEW_VALUES): Modifies the specified attribute of the file, assuming this modification is supported. For example, it does not appear necessary to change the replication factor of a file after it has been created.

DELETE_FILE (NAME): Removes the file specified by NAME.

6.3 File Access

These operations are required for the reading and writing of blocks of data to files.

FILEHANDLE: = OPEN (FILENAME, MODE, LOCK, TRANSACTIONID): Returns a file handle to be used by subsequent operations. The MODE parameter specifies read, write, append. It also specifies fixed-version or not. Fixed-version means that subsequent changes to the file (even committed changes) do not show up in the open file blocks. The LOCK parameter specifies whether the file is to be locked on open (read or write) plus the timeout period to wait for the file if the file is alread locked. The TRANSACTIONID parameter specifies the transaction with which this open file should be associated. This is used by the locking.

GET (FILE, BLOCK_NUMBER, BLOCKS, BUFFER, LOCK): Returns the specified number of blocks starting at BLOCK_NUMBER in the specified file. The LOCK parameter can specify no locking, read lock or write lock on the file plus a timeout period to wait for the blocks if there is already an incompatible lock on one or more of the blocks.

PUT (FILE, BLOCK_NUMBER, BLOCKS, BUFFER): Writes the specified number of blocks from BUFFER into the specified at BLOCK_NUMBER.

CLOSE (FILE, MODE): Flushs all writes to disk, invalidate the file handle and release OS resources associated with the file. The mode can specify "commit" or "abort". On commit the changes to the file are made atomically. On abort, the changes resulting from writing to this open file are discarded. In either case, all the locks associated with this open file are released.

SAVE_POINT (FILE, UNLOCK): Flushs all changes to the external file and set this point as the state to which the file will return on abort. The UNLOCK parameter specifies whether to release all the locks or not. Thus, it is like a close-commit without closing the file, i.e., losing access to the file.

6.4 Locking

Lock management provides locking of files as well as blocks within files. The lock management for a file resides in the file manager implementing the file. There is not a "floating" global lock manager. Co-locating locks with the file managers also allows us to combine lock operations with read operations, thereby reducing communication costs when operations are remote.

Lock modes include shared-read and exclusive-write. Provision should probably be made for more lock modes. A lock is associated with a particular transaction identifier and user account. For example, a write to a locked block fails unless the writing task specifies the same transaction identifier as recorded for the lock, plus has the same user account as the creator of the lock. The transaction association (versus a process/task association) allows multiple tasks to be involved in a transaction. The account association is for protection between users.

The following lock operations extract the transaction number from that specified in opening the specified open file. The user account is that associated with the requesting task.

> LOCK_FILE (FILE, LOCK): Sets the specified lock on this open file, waiting the timeout period if it is already locked in an incompatible mode.
>
> LOCK_BLOCK (FILE, BLOCK_NUMBER, LOCK): Locks the specified blocks in the open file, waiting the timeout period if it is already locked in an incompatible mode.
>
> UNLOCK_BLOCK (FILE, BLOCK_NUMBER): Releases the lock on the specified block. It would be nice to have this operation, as opposed to the full "open file" release of locks. However, it may be necessary for selectively releasing locks on data that is not needed in a transaction. Note: there is no individual UNLOCK_FILE (file) operation because it is provided effectively by the SAVE_POINT operation.

There is some desire to unlock all locks associated with a transaction. However, there seems to be no reasonable way for the operating system to have this information. The common mode of releasing multiple locks will be to use the multi-lock release of CLOSE and SAVE_POINT. The database system releases all locks associated with a transaction by closing the files associated with the transaction, or if it wishes to retain access to the files, using the SAVE_POINT operation.

It is necessary for file managers to have a reasonable degree of autonomy. A file manager is committing resources to a transaction in the form of open files and their associated open files. For this autonomy, it is necessary for a file manager to reclaim these resources under certain situations. This means that the file manager must be able to effectively abort the open file, releasing locks and undoing any changes resulting from this open file. This undo facility is clearly required or else file aborts of this nature would leave the file in an inconsistent state.

Thus, the file manager maintains an idle-time value associated with each open file. For open files that have been inactive for an extended period (or perhaps when there are significant lock requests), the file manager may contact the transaction manager about the state of this transaction. The transaction manager is a module that performs global deadlock detection and transaction registration, perhaps provided by the operating system. The transaction manager advises the file manager whether it may release the lock or not since it monitors the progress of transactions, particular checking for deadlock. The file manager can still (in critical situations) unilaterally decide to abort an open file but needs to so inform the transaction manager.

6.5 Backup and Recovery

The file managers must, for autonomy again, provide for backup and recovery independent of the DBMS. For instance, the DBMS log facility may be on another machine which is unable at a time required for critical recovery by a file manager. Also, the file managers must perform some sort of journaling anyway to support atomic transactions on files.

Shadow paging for atomic update is unacceptable in database files because of the negative influence on disk contiguity. Therefore, old pages must be logged when updated and restored from the log if the file is aborted.

The following might be a feasible way to run the log. Each time a page is updated, the page is updated in place plus a copy of the new page is written to the log with an indication of the file, time, page and transaction. Thus, the log contains the new version plus (from the last update) the previous version of the page. If an open file is aborted, the page is restored to its previous value from the log and the new page value is deleted from the log. There are two obvious optimizations. First, the new page is only tentatively written to the log until the file is committed, e.g., stored in a log staging area. This hopefully makes it easier to delete from the log when the transaction is invalid. Second, the old page may be written to the log as well if the cost of retrieving the old page has become too high. The page has not been updated for sometime and so the current page value is stored far back in the log. As a consequent of this logging, the DBMS may not need to log at all for recovery. However, it may still log at the field level (as opposed to the page level) for audit reasons. However, the reduced data rates for field level logging minimizes the cost of the redundant logging that this design leads to.

6.6 File Replication

A file may be defined to be replicated some number of times. The file replication is for survivability and performance. Therefore, this is specifying physically dispersed replication as opposed to disk mirroring or closely coupled replication on the same machine.

The DBMS must be able to specify/suggest the sites at which the files should be replicated so as to co-locate related data, e.g., relations that commonly are joined. There is some issue as to what the file system does when the suggested replication locations cannot be used. It may be simplest to return an error so the DBMS can pick an alternative or specify "don't care".

When a replicated file is accessed for reading, the operating system provides access to the replica with the least cost to access. The operating system may take into account communication speeds, processor load and other factors. The DBMS is expected to arrange for OPEN's to take place on the machine in which the data is to be used so that the rule used by the operating system leads to efficient DBMS operation.

There are at least two issues unresolved with read access to replicated files. First, should the operating system transparently switch access to a different replica of the file if access to the replica it chose at open time fails? Certainly --- doing so means knowing the locks that were set on the open file. This suggests that it may be much easier to just signal an exception on failure and not provide transparent recovery.

The second issue is whether some control is needed of which replica to use because of just the OS metric. For instance, suppose the route to one replica is known to be under attack or one is known to be safer in the current situation. This sounds fancy, but under duress the possibility is that the OS will pick precisely the wrong replica.

With writing to replicated files, there is a need to update all copies of course. Similarly, write locks have to be placed on all copies. This is an example of where multicast communication support could be very effectively used.

The major issue here is how to handle the case of not all copies being available. From an availability and survivability standpoint, it is infeasible to refuse service just because not all copies are available. The simple failure case to handle is when one or more replicas are not available and are known to be deadeither permanently or temporarily. In this case, we write to the copies that are available and provide a mechanism to update the other copies before they are made available in the system again.

Two issues place strain on this scenario. First, the replicated copies may be partitioned. That is, some copies may be available to us and some to others but not to both. Here seems best for the file system to indicate the problem to the DBMS (or application in general) but not prevent access. Future remerger of replicas may require human intervention but we judge that better than preventing operation altogether in partition state.

Second, it may be more important in some cases to provide access to a out-of-date replica than wait while it is brought up to date. The key issue here seems to be providing a means of doing with that minimizes the complexity of doing this since this is only likely to occur in infrequent situations of duress.

7.0 SECONDARY STORAGE MANAGEMENT MODULE

7.1 Objectives and Requirements

The main objective of this task is to design, implement, and evaluate a set of Ada packages that provides high-level support for secondary storage management for the WIS environment. This set of Ada packages is collectively called the Secondary Storage Management Module (SSMM) and must adhere to the CAIS standard node model [CAIS 85] and include discretionary and mandatory access control at the B3 level [NCSC 83]. It must also consider uniprocessors, multi-processors, and multi-computer systems.

The SSMM interacts directly with the WIS OS Kernel, the Program Execution Module (PEM), and the Transaction Manager (TM) of the DBMS to provide file and secondary storage service/support to WIS. It relies on a network-based File Server (FS) and a conventional Disk Server (DS) to perform network-wide file management and disk I/O, respectively. The SSMM also interacts with the Logging and Auditing Module (LAM) to provide security monitoring.

The FS must be designed to support well-known file access methods, virtual memory, and database applications [SVOB 84] by providing a simple interface conforming to CAIS. It must be based on a client-server model in,which multiple servers may cooperate to service a single transaction in an atomic fashion [STUR 80]. An atomic transaction may involve a single file or a set of files located across the network [FRID 81]. To support the illusion of a single, logical file system, a file-locating facility (naming and directory service) must be made available for providing file-location transparency to the user. For survivability and performance reasons, some files may be replicated, thereby requiring the maintenance of mutual consistency among the replicated files [WALK 83]. Locking and/or timestamp mechanisms must be provided for concurrency control [GRAY 79] [THOS 79] [BERN 81]. Appropriate deadlock control mechanisms must be provided to resolve any potential deadlock problems [ELMA 85]. Finally, facilities for automatic backup and recovery must exist to deal with system crashes and transaction abortions [HAER 83].

7.2 Adherence to CAIS

The CAIS model identifies three kinds of nodes: structural, file, and process nodes [CAIS 85]. A node may have contents, relationships, and attributes, but the contents vary with the kind of node. For a process node, the contents is a representation of the execution of an Ada program. For a file node, the contents is an Ada external file, which may represent a host file, a device (e.g., terminal or tape drive), or a queue (used for interprocess communication). For a structural node, there are no contents and the node is used strictly as a holder of relationships and attributes. The purpose of the node is solely to be a carrier of common information about other nodes related to it. Structural nodes are typically used to create conventional directories, configuration objects, etc.

Several predefined attributes are applicable to file nodes. The attribute FILE_KIND denotes the kind of file that is represented by the contents of the file nodes. The CAIS defines four kinds of files: SECONDARY_STORAGE, QUEUE, TERMINAL, and MAGNETIC_TAPE. These attribute values determine which of the ten CAIS I/O packages may be used to operate on files.

A secondary storage file in the CAIS represents a disk or other random-access storage file. Secondary storage files may be created and accessed by the use of CAIS sequential, direct, and text I/O packages. This is specified by the attribute ACCESS_METHOD.

Ada I/O involves the transfer of data to and from Ada external files using predefined packages. CAIS I/O uses the same approach and involves the transfer of data to and from the contents of CAIS file nodes. Like Ada I/O packages, however, the operations in the CAIS I/O packages are expressed as operations on objects of some file types (in internal files), rather than directly on the external files. These objects are internal (files) to a CAIS process and are identified by file handles. Note: in the context of the CAIS, the word file is used to mean an Ada external file, whereas in the context of the Ada Language Reference Manual (LRM) [1815A 83], file is used to mean an internal file.

Of the ten predefined CAIS I/O packages, the SSMM must implement the following:

a. CAIS.IO_DEFINITION: Defines the types and exception associated with file nodes.

b. CAIS.IO_CONTROL: Defines facilities that may be used to modify or query the fundamentally of CAIS files.

c. CAIS.DIRECT_IO: Provides facilities for accessing direct files; comparable to those facilities described on the DIRECT_IO package of the Ada LRM [1815A 83].

d. CAIS.SEQUENTIAL_IO: Provides facilities for accessing sequential files; comparable to those facilities described in SEQUENTIAL_IO package of the Ada LRM [1815A 83].

e. CAIS.TEXT_IO: Provides facilities for accessing text files; comparable to those facilities described in the TEXT_IO package of the Ada LRM.

f. CAIS.FILE_IMPORT_EXPORT: Allows a particular CAIS implementation to maintain files separately from those maintained by the host file system.

To enhance portability the CAIS model also specifies minimum values for implementation-determined quantities and sizes. Those which apply to file nodes include the maximum record size (which must be at least 64K bits) and the range of indexes for a direct-access file (which must be at least from 1to 64K).

7.3 Basic Model

A file system consists of software that manages permanent data objects (in the sense that their values persist longer than the processes that create and use them). They are kept in files on secondary storage devices like disks. These files are organized into a tree-structured directory. Conceptually, each file consists of a sequence of data objects. Thefile system provides basic operations that create or delete a file, open a file given its name, read the next object for an open file, write an object onto an open file, or close a file.

A file system is that part of the operating system that implements files stored in secondary-storage devices like disks. A file system has two basically different functions: (1) it is responsible for manipulating files in an efficient manner; and (2) it gives the user the ability to create his/her own name space and to store and retrieve data from it in a flexible way.

The disk hardware has no built-in notion of files or directories. It is the file system software that provides such facilities. At the disk driver level, a disk is nothing but a large array of data blocks that can be accessed randomly using three basic operations: select a block (seek), copy the contents of the selected block from disk to memory (read), or copy the contents of memory to the selected block (write). The design of the DS is conventional in nature and should adhere to the common interface requirements for I/O drivers.

A file is a sequence of bytes and its physical implementation may use fixed-size blocks (which is a low-level decision and not visible to the user). The DBMS should probably set a fixed-size block, for example, one KB or larger. For compatibility with other systems, variable-size blocks may be required. Both sequential and random access should be allowed.

A file system establishes a name within a context by constructing directories. The user refers to a specific file by a symbolic name. For each symbolic name in a given context, there is a directory entry that translates the symbolic name into the actual location where the file resides. This translation may be effected by going through more than one level of directories.

Thus, in addition to conforming to the CAIS requirements, the basic file system must incorporate the following characteristics into its design:

a. The ability to allocate secondary storage effectively

b. The ability to access files by names

c. The ability to share common files

d. Flexiblility and versatility of access

e. Security and integrity of the information stored in files

f. Efficient implementation of file manipulation commands

7.4 File Management

The basic file system must provide file management operations to create or delete a file, to open or close a file given its names, and to reset the current index to the beginning of a file. These operations are specified as appropriate procedures in CAIS.DIRECT_IO, CAIS.SEQUENTIAL_IO and CAIS.TEXT_IO packages. In the following discussion, the parameters of these procedures are not listed due to their length and because they are listed fully in the CAIS document [CAIS 85]. However, standard parameters should be used, where appropriate, for each of these procedures (and functions as well).

As an illustration of the interface to these procedures, the first procedure's interface (i.e., create a direct input or output file) is given below.

CREATE a direct input or output file.

```
procedure CREATE (FILE:   in out FILE_TYPE;
    BASE:                 in  NODE_TYPE;
    KEY:                  in  RELATIONSHIP_KEY := LATEST_KEY;
    RELATION:             in  RELATION_NAME := DEFAULT_RELATION;
    MODE:                 in  FILE_MODE := INPUT_FILE;
    FORM:                 in  LIST_TYPE := EMPTY_LIST;
    ATTRIBUTE:            in  LIST_TYPE := EMPTY_LIST;
    ACCESS_CONTROL:       in  LIST_TYPE := EMPTY_LIST;
    LEVEL:                in  LIST_TYPE := EMPTY_LIST);
```

This procedure creates a new (external) direct input or output file and its file node; each element of the file is directly addressable by an index. FILE describes the type for the file handle (i.e., internal file) for all direct I/O operations. MODE indicates whether input operations, output operations or both can be performed on the direct-access file. The FORM parameter is used to provide file characteristics concerning the creation of the file, such as file name and size. The ATTRIBUTE parameter defines and provides initial values for attributes of the file. The ACCESS_CONTROL parameter specifies initial access control information to be established for the created node. Finally, the LEVEL parameter specifies the security level at which the file node is to be created.

OPEN a direct input or output file. This procedure opens a file handle on a file, given an open node handle to the file node (i.e., associates the given internal file with an existing external file having the given name and form).

CLOSE a direct input or output file. This procedure closes a file handle on a file (i.e., severs the association between the given internal file and its associated external file).

DELETE a direct input or output file. This procedure deletes the (external) file associated with the given file handle; the internal file is closed, and the external file ceases to exist.

RESET a direct input or output file. This procedure resets the given file handle so that reading from or writing into its elements can be restarted from the beginning of the internal file.

The above five file management operations also apply to SEQUENTIAL_IO, except that the attribute ACCESS_METHOD is assigned the value SEQUENTIAL as part of the creation. For TEXT_IO, the procedures CREATE, OPEN, and RESET have additional effects; see the CAIS document [CAIS 85] for details.

7.5 File Access

In addition to the five file management operations described in Section 7.4, the basic file system must implement the following file access operations as specified in the CAIS I/O packages:

a. CAIS.DIRECT_IO implements READ, WRITE, AND SET_INDEX.

b. CAIS.SEQUENTIAL_IO implements READ and WRITE.

c. CAIS.TEXT_IO implements GET, PUT, SET_INPUT, SET_OUTPUT, SET_LINE_LENGTH, NEW_LINE, SKIP_LINE, NEW_PAGE, SKIP_PAGE, SET_COL, and SET_LINE

Note that the package CAIS.TEXT_IO provides facilities for input and output in human-readable form (i.e., textual data). It also provides facilities for default input and output (i.e., standard input and output) file manipulation, and for layout-control operation. Each textual file is read (GET) or written (PUT) sequentially, as a sequence of characters grouped into lines, and as a sequence of lines grouped into pages.

In summary, much of the support needed for the basic file system is available from the CAIS I/O packages. The SSMM needs to implement the procedures and functions of these packages to provide basic file and secondary-storage services to WIS in a single-host environment. However, in a multiple-host local-network environment, additional packages must be developed; this is discussed in the following section.

7.6 Network-Based File Server

The network-based File Server (FS) [SVOB 84] must do more than the basic file system, as described in the previous section (i.e., more than the conventional manipulation of individual files on the secondary storage). It must provide an abstract name space and high-level operations to manipulate objects in that space. Names in the abstract space may refer to devices, services that the system supplies, or files that reside on other sites. It must also provide mechanisms needed for cooperating with the TM to support both basic and nested transactions in a secure, reliable and distributed environment of WIS. Examples of these mechanisms are those used for replication, concurrency control, deadlock control, and automatic backup and recovery. Note: research is required to develop some of these mechanisms; the specific mechanisms chosen.

7.7 Naming and Directory

In order to provide file location transparency to the user, a Name Server that allows the user to access a file by name rather than by location should be available to the system. The CAIS package NODE_MANAGEMENT may be used to locate a file node.

For each file in the system, an entry must be kept in a directory. In addition to the file's name, owner, physical and logical structure that are needed in the basic file system, the directory entry should include the file's other attributes that are needed by the TM for replication, concurrency control, deadlock control, and backup and recovery. Moreover, attributes that are needed by the Logging and Auditing Module must also be included, e.g., time of last reference, last access, security level, link count, etc. In the CAIS model, a directory is represented by a structural node, which can be created by using the STRUCTURAL_NODES package. Setting and changing of file attributes and access control information can be done by invoking the ATTRIBUTES package and the ACCESS_CONTROL package, respectively.

The collecuon of the directory entries for a number of files may in turn form a higher-level directory, resulting in a hierarchical directory system. For efficiency and reliability reasons, a directory may be fully or partially replicated at different sites. Copying a structural node and a tree of structural nodes can be done in CAIS by invoking procedures COPY_NODE and COPY_TREE, respectively, in the N0DE_MANAGEMENT package.

7.8 Replication

Replication of secondary storage files in a distributed file system serves multiple purposes. In order to increase survivability and enhance reliability, it is desirable to support the replication of critical data elements at multiple sites. Since replication increases the cost of update but improves retrieval dramatically, data elements that are often read but infrequently updated should be replicated at sites where the queries originate. Copying a file node can be done in CAIS by invoking procedure COPY_NODE in the NODE_MANAGEMENT package.

In general, the system should provide the support of replication transparency to the user. That is, all details of locating and maintaining replicas should be handled by the system, not by the user. The contractor is required to provide a policy module that specifies when, where, and how replication takes place, and to develop and justify algorithms for the policy module. Moreover, provisions should also be made available to the user for specifying the degree of replication and for suggesting replication sites, if the user so desires.

7.9 Concurrency Control

Concurrent execution of different transactions may result in one transaction's observation of an inconsistent, transient state created by another transaction during its execution. Concurrency control deals with ensuring transaction atomicity in the presence of concurrent execution of transactions. It is usedby the system to prevent updates performed by one user from interfering with retrievals and upda.es performed by another. During the past decade, scores of concurrency control algorithms have been proposed in the literature; however, they can be classified into two basic classes: two-phase locking and timestamp ordering [BERN 81]. A third class, called optimistic methods, essentially uses timestamp ordering; therefore, it can be treated as such, as far as low-level mechanisms are concerned.

Of these two techniques, locking is currently more popular and has been used by several commercial database systems. However, there is some feeling among researchers that timestamp ordering may be more appropriate for distributed systems like WIS, and it will become more widely used in the future. It is recommended that in its initial implemention the popular locking approach be used for concurrency control.

A concurrency control mechanism (or mechanisms) should be specified and its choice justified. It should also determine all implications of such a choice on the system, and provide for support of the mechanism whenever needed.

Lock management should provide locking of files and records within a file. Lock modes should include shared-read and exclusive-write, and provisions should be made for more lock modes (such as Read Copy and Write Copy [FRID 81]), if appropriate.

A timestamp can be obtained through a service call to the Time Synchronization Agent. Each transaction must receive a timestamp when it is initiated at the originating site, and each read or write operation required by a transaction must have the timestamp of the transaction. The system must maintain these timestamps for use by the concurrency control algorithms until the transaction is successfully completed.

7.10 Deadlock Control

Deadlock is a situation in which two or more transactions are in a simultaneous wait state, each waiting for one of the others to release a lock before it can proceed further. One of the simplest and most widely used methods to resolve this kind of deadlock is to specify a maximum wait time (timeout interval), and roll back the transaction which is waiting if the time expires before the request is granted. The main problem with this method is to determine a good timeout interval. If it is too short, more transactions which are not in deadlock may be unnecessarily aborted; if it is too long, more time will be wasted by transactions in deadlock before being aborted. In distributed systems like WIS, it is even harder to determine a workable timeout interval, since the behavior of the communications network and of the remote sites involved in executing a transaction is less predictable. Other well-known, undesirable side effects of the timeout method are <u>cascading aborts</u> and <u>lost updates</u>.

Other principal mechanisms commonly used to solve the deadlock problem are explicit deadlock detection and <u>a prior</u> ordering of transactions based on timestamps. The specific mechanism specified for deadlock control should be justified. It is recommended that fault tolerance be incorporated into the mechanism used [ELMA 85].

7.11 Backup and Recovery

Nothing will ever work 100 percent of the time. Accordingly, the FS must incorporate not only a variety of checks and controls to reduce the likelihood of failures, but also a set of procedures and mechanisms for recovering from the failures should they occur during the execution of transactions. There are three types of failure that are commonly found in a database system: (1) transaction failure, (2) system failure, and (3) media failure [HAER 83]. (All communication network failures will be taken care of by the Network Protocols Task Force Group.)

Transaction failure is a failure caused by unplanned, abnormal program termination, such as arithmetic overflow, division by zero, consistency violation, deadlock timeout, protection violation, and the like.

Transaction failure means that the transaction has not reached its planned termination point (commit). Thus, it is necessary for the system to undo any changes that the transaction has made to the database. Undoing changes involves working backward through the log, tracing through all log records for the transaction until its beginning point is reached. For each log record encountered, the change represented by that log record is undone by replacing the new value in the database by the old value from the log.

System failure is a failure caused by a bug in the DBMS code, an OS fault, or a hardware failure. It causes the system to stop and thus requires a subsequent system restart. The contents of main memory are lost, but the database is not damaged. The notions of checkpoint, undo/redo logic, and write-ahead log protocol are commonly used to help recover from system failure.

Media failure is a failure in which some portion of the secondary storage medium is damaged. It may be caused by bugs in the OS routines for writing the disk, hardware errors in the channel or disk controller, a head crash, or a loss of information due to magnetic decay. The recovery process in this case consists of restoring the database from a backup copy, using log to redo the transactions executed since that backup copy (dump) was taken.

The basic technique used to ensure file recoverability is the shadow-page technique, which avoids overwriting the actual data in the physical storage until after the commitment. An alternative technique is to update files in place, with the help of an undo/redo log. Another technique is the use of multiple versions [SVOB 84], which creates a tentative version of the whole file for a write-data request. Recovery may be performed either immediatelyafter the crash or completed as needed. The CAIS IO_CONTROL package provides facilities for handling log files and the NODE_MANAGEMENT package provides facilities for copying files. The specific mechanisms used for performing automatic backup and recovery.

8.0 TRANSACTION MANAGEMENT

The atomic transaction management facility provides a coordination and logging facility for transactions. This facility is not considered in either the Ada language specification or in the CAIS specifications. Therefore, the design is provided purely as a compatible extension of the CAIS standard.

An atomic transaction is a series of operations parenthesized by begin-transaction and end-transaction statements with provision for making these operations atomic. Either all the operations are performed and are performed indivisibly with respect to other clients, transactions, operations and failures or else none of the operations is performed. In the latter case, the transaction is said to have been aborted.

A transaction is more commonly a sequence of file or database operations. However, the actions or operations of any service can be included in a transaction provided that it supports the transaction protocol.

Transactions may be nested. In this case, a subtransaction is carried out relative to its parent transaction. An abort of the subtransaction only undoes the actions of that subtransaction (and any of its subtransactions). A commit of a subtransaction only makes the changes performed by the transaction visible to the actions contained in the parent transaction and only commits these changes relative to the parent transaction. In particular, aborting the parent transaction undoes changes to its subtransactions even if those subtransactions have been committed. In this sense, committing a subtransaction effectively merges its actions and the "fate" of its actions with the parent transaction.

The WIS OS transaction management systems consists of three major pieces: a transaction manager, a transaction protocol and client transaction software. In addition, servers include software to support the transaction protocol, as required. As mentioned above, any server that does implement that transaction protocol can participate in a transaction.

8.1 Client Transaction Software

An Ada package implements the client interface to the system transaction management. This package creates a transaction record for each transaction the client starts and provides entry procedures for the client to manipulate transaction records.

The following entry procedures are suggested.

> CREATE_TRANSACTION (SUPERTRANSACTION, TRANSACTRECORD): Creates a new transaction record and register the new transaction with the transaction manager. The transaction is created as a subtransaction of SUPERTRANSACTION if this parameter is non-null.
>
> ADD_SERVER_TO_TRANSACTION (TRANSACTRECORD, SERVER_TASK): Adds this server as a module that is performing actions to this transaction. The server task is then informed of transaction management operations on the transaction (commit, abort, prepare-to-commit, etc) as well as recorded in the transaction log as a participant in this transaction. A server task is not added if it is already recorded as associated with this transaction.
>
> ABORT_TRANSACTION (TRANSACTRECORD): Terminates the specified transaction, informing the transaction manager and all the service modules that are participating in the transaction of the action. This operation may be performed at

any time up to the commitment of the transaction. The transaction record is freed after the transaction is aborted.

PREPARE_TO_COMMIT (TRANSACTRECORD): Gets agreement from all server modules in the transaction that they are prepared to commit the transaction. An error return is given if all server modules cannot be reached or do not agree to commit. There is no general recourse after such an error return than to abort the transaction.

COMMIT_TRANSACTION (TRANSACTRECORD): Completes the transaction by informing the transaction manager and all the participating servers of the decision to commit. The transaction record is deleted.

COMMIT_CONTINUE_TRANSACTION (TRANSACTRECORD): The transaction changes are commited as with COMMIT_TRANSACTION. However, the transaction continues in existence with no locks released. In particular, the state of the transaction after a successful COMMIT_CONTINUE _TRANSACTION operation is "in-progress". This effectively allows an intermediate commitment as a transaction savepoint.

In addition, there should be programs for querying the transaction log of completed transactions as well as querying the state of current transactions.

8.2 Atomic Transaction Protocol

An atomic transaction is a grouping of actions or operations to be atomic or indivisible with respect to other operations and failure. The atomicity of a sequence of actions distributed across multiple different machines is accomplished by coordination at three levels. First, each server coordinates the actions local to it into an atomic transaction. Second, the client and servers are mutually coordinated to make the collection of local actions "committable" as an atomic transaction. Finally, the client coordinates with the transaction manager to ensure that a copy of the transaction commitment or abort is made in stable storage. This coordination is accomplished by the atomic transaction protocol.

The transaction protocol is implemented by the client, the servers and the transaction manager. The transaction manager provides reliable storage for recording the current state of each current transaction as well as for storing the final result (commit or abort) for each transaction. Thus, a server or client sends to the transaction manager requesting the current state of a given transaction, determining whether it is in progress, aborted, committed, etc. A server can also unilaterally abort a transaction providing the server has not already agreed to commit the transaction. Each server implements atomic transactions for the operations it performs. Finally, the client actually drives the transaction, requesting the operations that constitute the transaction and deciding on whether to commit or abort after performing the actions.

A particular aspect of this design is the reliance on the client to carry out the transaction protocol correctly. Otherwise, the transaction may be carried out incorrectly. For instance, part of the transaction in one server may be commited and another part aborted. This client dependence is considered acceptable because of the following:

a. A simple failure of the client, such as hardware crash, does not result in an incorrect transaction.

b. The correctness of the transaction is dependent on the correctness of the client-invoked actions as part of the transaction anyway. Having the transaction management dependent on client behavior does not decrease the reliability of the transaction mechanism.

The protocol is described next focusing on the client messages sent to the transaction manager and how it should respond. Similarly for the participating servers.

A transaction is identified by a transaction identifier, or trans-id for brevity. We propose that task group identifiers (see Section 3.0) be used as transaction identifiers. This has two advantages:

a. The task group mechanism guarantees that the identifier is unique in the system, a requirement for transaction identifiers.

b. The group operations on tasks can be used to efficiently communicate aborts, commits and prepare-to-commits from the client. Having the transaction identifier as a group identifier allows the servers to know the group identifier with no extra effort.

The primary disadvantage is the resulting restriction on the number of transaction identifiers available, therefore requiring that transaction identifiers be recycled over time. To deal with recycled transaction identifiers, a timestamp is associated with each transaction. Thus, the full transaction identifier is (timestamp, task-group-id), which is unique forever. The transaction manager guarantees that task-group-id's are not reused with less than some minimum recycle time T. Thus, a transaction can be uniquely identified by the task-group-id and a time that the transaction existed. Assuming each client and server has a reasonably accurate clock (within T/2 of the transaction manager clock) and remembers when it first encountered a particular transaction, it can uniquely identify the transaction at any point in the future. Since the task-group-id is unique while the transaction is active, there is no need to carry the timestamp in most transaction messages. In fact, it is only used in query, prepare-to-commit, commit, and abort operations.

8.2.1 Transaction Manager Protocol

The transaction manager is a logically central repository for information regarding the current state of transactions. The client invokes entry points in the transaction manager as remote procedure calls, as normal. These calls are hidden in the client library routines. They are described here to clearly delineate the local client software from the global server software. This partitioning is a key issue in distributed systems design. The protocol with the transaction manager, in terms of these entry points, is as follows.

A client library, as part of CREATE_TRANSACTION, invokes the

```
ALLOCATE_TRANS (SUPERTRANSID, GROUP_TASKID, TIME)
```

to get the transaction manager to allocate a unique GROUP_TASKID and returnthis value (in SUPERTRANSID) plus a timestamp for the transaction. Again, (TIME, GROUP_TASKID) is a transaction identifier for this transaction. The transaction manager retains a record, indicating that this transaction is "in-progress".

A server can invoke

```
QUERY_TRANS (GROUP_TASKID, TIME, STATUS)
```

to determine the status of the transaction, returned in STATUS. The TIME parameter need not match exactly the timestamp used by the transaction manager. It need only be within error tolerances of the transaction manager for the matching on GROUP_TASKID to succeed. Typically, the TIME parameter is set to the time the server first encountered transaction. Again, the recycling of group-task-id's is controlled such that there is no ambiguity in this matching within the tolerances of the server clocks.

The client or a server can invoke

```
ABORT_TRANS (TRANSID, TRANSRECORD)
```

to terminate the transaction. The client can provide in the parameter TRANSRECORD a transaction record to be stored in the transaction manager's log. This is for auditing purposes only. All the other servers are informed by the task group communication mechanism. It may be useful to raise an exception in the client task when an abort is caused by a server in the transaction. Responding to this call, the transaction manager writes the specified transaction recordto its log file, including the final status as aborted. It then deletes its active record for the transaction.

Queries for this transaction subsequently cause the transaction manager to refer to its log for the information. The client can invoke

```
COMMIT_TRANS (TRANSID, TRANS-RECORD, CONTINUE_TIME)
```

which marks the transaction as committed, stores the transaction record specified by TRANSRECORD in the transaction manager log, and deletes the active record for this transaction. Again, only the final status of the transaction is critical for the correct operation of the protocol. The full transaction record is logged for audit purposes. If the CONTINUE_TIME parameter is non-zero, the transaction is commited as a savepoint but continues as in-progress with a new timestamp (but same task-group-id) which is returned in this same parameter.

Notice that the transaction manager does not know the service modules involved with the transaction until the commit or abort occurs. It only knows whether the transaction is active or not. In this vein, the transaction manager associates an owner task with a transaction. If this owner task is deemed to no longer exist, the transaction manager is free to abort the transaction, thereby garbage collecting transactions for both itself and the associated servers.

8.2.2 Server Interface

A transaction identifier parameter is provided in entry procedures to the various service modules. For example,

```
OPEN (NODE, NAME, INTENT, TIME_LIMIT, TRANSID)
```

Also, operations like GET and PUT are associated with a transaction by virtue of the OPEN or CREATE operation being tiedto a transaction. That is, there is no need for an extra parameter on these operations. In any case, operations within a service module are associated with a transaction in either of these two ways. When a server task becomes involved with a transaction for the first time, it joins the task group associated with the transaction. This allows it to receive all subsequent transaction management operations.

The client and server are assumed to take the necessary actions with locking to ensure proper concurrency control. In response to a PREPARE_TO_COMMIT_TRANS invocation, the server ensures that all actions associated with this transaction can be committed, updates its timestamp from that supplied in the message and responds with a vote - commit or no commit. If a server has any problem with committing the transaction operations, it must vote no. Servers whose only actions have been only reads (with no modifications) can vote yes and disconnect themselves from the transaction unless the continue flag is true. (This is because an abort and commit are identical for these servers.) The updated timestamp allows servers to properly identify the portion of a transaction that occurs before a commit-continue operation versus those that occur after the commit-continue. That is, the transaction up to the commit-continue is identified with the specified timestamp. The continued transaction is identified by the timestamp returned by the transaction manager in response to the commit operation.

To complete or savepoint the transaction, the client invokes

COMMIT_TRANS (TRANSID, TIMESTAMP, PARENTTRANS, CONTINUE)

at the servers in the transaction. The TRANSID and TIMESTAMP parameters must match that used in the PREPARE_TO_ COMMIT _TRANS operations. In response, a server commits its portion of the transaction actions and releases the resources it has associated with the transaction, assuming CONTINUE is false. These "resources" include any locks associated with the transaction.

Similarly,

ABORT_TRANS (TRANSID, TIMESTAMP)

causes the server to abort the operations it has performed as part of the transaction. In this case, the server undoes all actions performed as part of the transaction. If a server needs confirmation of the status of a transaction (such as when it is waiting commit after a prepare-to-commit), it may invoke

QUERY_TRANS (TRANSID, TIMESTAMP, STATUS, PARENTTRANS)

which returns the status of the transaction in the so-named parameter plus the parent transaction, if any. This procedure is implemented by the transaction manager. In addition, a server can unilaterally abort a transaction that it has not agreed to commit (in response to a PREPARE_TO_COMMIT) by invoking

SERVER_ABORT_TRANS (TRANSID, TIMESTAMP, TRANSACTRECORD)

which is implemented by the transaction manager. The TRANSACTRECORD parameter is for logging and audit purposes only and (optionally) describes the server's involvement in the transaction and its reason for issuing the abort. In response, the transaction manager invokes an abort operation in the other servers using the trans-id as a task group identifier to reach every server in the transaction.

As part of a transaction, a server may be asked to create lock and access objects that are already locked by another transaction. The server can invoke

QUERY_TRANS_ RELATION (TRANSID, TIMESTAMP, TRANSID2, TIMESTAMP2, RESULT) to determine the relation, if any, between two transactions. This operation is implemented by the transaction manager. If the requesting transaction is a subtransaction (directly or indirectly) of the transaction holding the locks, locking and access is allowed. Otherwise, the request is refused.

In addition, a server may be called upon to use an existing object in a subtransaction of that originally using it. For example, a subtransaction may be passed an open file to update. Servers should support

```
CREATE_FILE_VERSION (NODE_HANDLE, TRANSACTION, PARENTTRANS)
```

which creates a new version of the open file for updates in the subtransaction. This allows the subtransaction to be aborted and undo only those changes to the open file that resulted from this transaction. The open file version is discarded, thereby returning to the state of the open file at the beginning of the transaction. When a transaction containing this created version is commited, the changes in the version are reflected in the object from which the version was created. This may in fact be another version. The "version depth" that a server need support is at most the maximum nesting of transactions, not the total number of subtransactions. Proposed is that each server support at least depth 3 although greater depth is preferred.

Finally, servers should support entry points that return information describing the transactions they are involved with and blocking on locks associated with these transactions. One use for these entry points is deadlock detection as described below.

8.3 Deadlock Handling

Every server should provide timeouts on locks so that deadlocks are eventually resolved. However, in this environment, timeouts may need to be so long to avoid random lock timeouts that it may be necessary to do deadlock detection explicitly. However, it should be replicated for reliability. Proposed is that the multiple instantiations of the transaction manager implement the transaction log as a replicated file using some replicated file update algorithm, such as the Gifford majority voting scheme, which allows transactions to continue even when one or more (but a minority) of the transaction managers are unavailable. When a transaction manager recovers, it must go through a recovery procedure that brings its log up to date with the current transaction log. In this vein, the transaction manager cannot acknowledge a commit or an abort request from the client until a majority of the transaction managers have agreed to, and stored a record of, this decision.

The transaction manager should be coded as an Ada module. It could reasonably be included as part of the network file servers, i.e., one transaction manager instantiation per file server.

In the proposed prototype implementation, transaction managers in different clusters operate independently. In particular, there is no guarantee that transaction identifiers in different clusters are unique as described. However, assuming there is some form of cluster identifier, the transaction identifier (cluster-id, timestamp, task-group-id) is unique across the entire system. When a client includes a server in another cluster in a transaction (thus going through the cluster gateway and creating an alias task), the server notes that the client is on a remote cluster and deals with the transaction manager on that remote cluster. That is, the server associated an object in a transaction (such as an open file) with a client, who is associated with a cluster. The transaction manager associated with this transaction is the transaction manager associated with the client's cluster.

The independence of the transaction managers in this design preserves the independence of clusters and avoids the overhead that would be associated with coordinating the transaction managers. The hierarchical structuring of the transaction identifier means that this is a system-wide unique identifier for all transactions, yet, only a very short form of this identifier is used with time-critical operations such as OPEN.

8.4 Server Issues

The mechanism described is primarily for coordinating multi-server transactions. In a system with a single server that could be involved in transactions, there would be no need for a separate transaction manager of the type described here. The protocol would be strictly between the clients and this one server. Thus, the transaction management does not reduce the server support required for atomic transactions. It merely specifies its interface.

Each server must allow its operations to be associated with a transaction with a mechanism for concurrency control and recovery to ensure actions local to a server can be made atomic. Thus, each server must provide a local lock manager module to control concurrency access. The lock manager should implement EXCLUSIVE_WRITE and EXCLUSIVE_READ, as specified in the CAIS. Page-level locking is also required.

For instance, a database service should allow update operations to be associated with a transaction. If the transaction is aborted, any such updates must be undone. Once the server agrees to a commitment of the transaction, it must be prepared to ensure the updates take place if the transaction is committed, independent of failures of either themselves or the transaction manager (latter should be unlikely). This means that each such server must have a stable storage log for updates associated with a transaction. Also, once a server has responded positively to a prepare-to-commit request for a transaction, it must not unilaterally abort or commit the transaction until the decision has been communicated either by the client or the transaction manager. In the worst case, various server resources may be locked and unavailable to other clients if the transaction manager is unavailable. Thus, the transaction manager must be quite reliable.

The major problem is expected to arise with remote cluster transactions and the reliability of communication between clusters. In particular, a server may be unable to release resources tied to a transaction originating in another cluster because communication with that cluster has failed between prepare-to-commit and commit.

8.5 Summary

The chapter describes a transaction management system for WIS, providing for distributed, multi-server, atomic transactions. The emphasis is on defining a standard interface to transaction management for clients and servers, not the actual mechanism. In particular, any server can participate in a transaction providing it implements the server transaction management interface. Similarly, the client can use the client transaction facility for any sequence of actions, providing they are implemented by servers that support the transaction interface.

9.0 PROGRAM EXECUTION MODULE

9.1 Objective

The main objective of this section is to design, implement, and evaluate a set of Ada packages that provides high level support for program execution for the WIS environment. The set of Ada packages that implement program execution support must adhere to the CAIS standard "node" model [CAIS 85], include discretionary and mandatory access control at the B3 level [NCSC 83], and consider uniprocessors, multi-processors, and multi-computer systems. While the WIS environment is a collection of local area networks connected by one or more wide area networks, the work being requested here concentrates on a local area network. The design and implementation should not preclude, nor make difficult, interactions with other local area networks of WIS. WIS is a demanding environment that must have the following attributes: fault tolerance, survivability, multi-level security at the B3 level, portability of applications and system software across a wide variety of machine sizes and types, single and multi-thread machines, multiple priorities, real-time processing, and fully integrated databases [JACK 84]. These requirements must be adhered to when designing and implementing the packages to support program execution. This gives rise to a number of interesting issues that must be resolved, including distributed resource management, distributed scheduling, deadlock resolution, and how to support the above list of attributes within the context of program execution.

9.2 Discussion

Program execution support is a collection of Ada packages supporting the execution of Ada programs at a high level and is called the Program Execution Module (PEM). The PEM is composed of two main parts: 1) the direct implementation of the CAIS PROCESS_CONTROL package, and 2) the implementation of scheduling algorithms which is itself composed of a local scheduler, a global scheduler, and a statistics package. The PEM relies on the kernel to provide execution support at a low level, e.g., see kernel primitives such as CREATE_PROCESS, DESTROY_TASK, SET_TASK_PRIORITY, etc. The kernel also contains a priority based dispatcher.

The main interfaces of interest for the PEM are the command language (CL), the kernel, and the secondary storage management module (SSMM). The CL is sophisticated and can support invoking typical functions such as compiling, linking, loading, and executing programs, as well as creating, deleting and manipulating files and libraries, etc. The kernel supports the compiling, linking and loading aspects of program execution, as well as providing virtual memory. The kernel and secondary storage management modules support the idea that all resources are "objects" so that the PEM can issue "GET OBJECT" calls for acquiring resources such as main memory and files. For fault tolerance requirements, objects may be replicated. Access to objects and maintaining consistency of objects is supported by the secondary storage management module.

The remainder of the discussion is about the PEM itself and is divided into four sections: adherence of the PEM to the CAIS, local scheduling, global resource management, and the collection of statistics. Also included is a brief note on deadlock resolution.

9.2.1 Adherence to the CAIS

The Common APSE Interface set (CAIS) has been developed to facilitate interoperability and transportability of tools and data between APSE's. The CAIS defines those interfaces most commonly required by tools in their normal operation. The scope of the CAIS includes interfaces traditionally provided by an operating system. Ideally, all APSE tools would be implemented using only the Ada language and the CAIS. Hence, this proposed project, together with the collection of other OS Task Force defined projects listed in the introduction, should develop a distributed operating system that supports the interfaces defined by the CAIS model. It does not mean that the full CAIS itself is necessarily implemented. However, after implementing the PEM, SSMM, and the kernel, much of the CAIS would be implemented or easily added.

The main ingredients of th CAIS model are discussed first, followed by a discussion of those aspects of the CAIS most closely related to the PEM. [CAIS 85] contains a more complete description of the CAIS model.

The CAIS model supports the concepts of nodes, relationships, and attributes. A node is a carrier of information about an entity. A relationship represents an interrelation between two entities. An attribute is the property of an entity or of an interrelation. The CAIS model identifies three types of nodes: structural, file and process nodes. A structural node contains relationships and attributes. In the proposed distributed operating system, the structural node is largely supported by the kernel, the secondary storage memory management module, and the authentication server (reference monitor). File nodes contain Ada external files. The file node is largely supported by the secondary storage management modules and the kernel. A process node has contents, relationships and attributes. The contents of a process node is called the process which represents the execution of an Ada program. A process node's attributes and relationships are used to bind the resources required by the process to the execution. Each time execution of a program is initiated the following occurs:

a. A process node is created.

b. The process is created.

c. The necessary resources to support the execution of the program are allocated to the process.

d. Execution is started.

In the CAIS model, a process node represents a single Ada program, even when that program includes multiple tasks. There is no requirement (other than those imposed by the semantics of the Ada language) on how these tasks are to be scheduled for execution. In this design WIS OS performs the scheduling of processes and tasks.

In addition, the CAIS has facilities for creating new processes in a hierarchical structure, i.e., the procedure. An entire tree of processes created in this way is called a job. A user may have multiple jobs executing in parallel, within each job multiple processes may be executing in parallel, and within each process multiple Ada tasks might be executing in parallel. The scheduling discipline must be defined and implemented. If a parent process terminates or aborts, or if a parent process suspends or resumes, all children processes follow suit.

In terms of the CAIS, the package PROCESS_CONTROL is closely related to program execution support. Therefore, this package must be implemented. The PROCESS_CONTROL package of CAIS defines 17 procedures and/or functions, describing their interface and functionality. In most cases these 17 procedures and/or functions are supported by the WIS OS kernel. The term "support" means that primitives are provided which can be used to implement these routines. The parameters of each of these procedures are not listed here because of their length and because they are fully listed in the CAIS document [CAIS 85]. However, the standard parameters should be used for each of these procedures and functions. As an illustration of the interface to these procedures, the first procedure's interface is described (i.e., the SPAWN_PROCESS).

SPAWN_PROCESS: Creates a new process as a child of the current process; supported by the kernel.

```
procedure SPAWN_PROCESS (
        NODE:                   in out  NODE_TYPE;
        FILE_NODE:              in  NODE_TYPE;
        INPUT_PARAMETERS:       in  PARAMETER_LIST := EMPTY_LIST;
        KEY:                    in  RELATIONSHIP_KEY := LATEST_KEY;
        RELATION:               in  RELATION_NAME:= DEFAULT_RELATION;
        ACCESS_CONTROL:         in  LIST_TYPE := EMPTY_LIST;
        LEVEL:                  in  LIST_TYPE:= EMPTY_LIST;
        ATTRIBUTES:             in  LIST_TYPE:= EMPTY_LIST;
        INPUT_FILE:             n   NAME_STRING:= CURRENT_INPUT;
        OUTPUT_FILE:            in  NAME_STRING:= CURRENT_OUTPUT;
        ERROR_FILE:             n   NAME_STRING:= CURRENT_ERROR;
        ENVIRONMENT_NODE:       in  NAME_STRING:=CURRENT_NODE;)
```

AWAIT_PROCESS_COMPLETION: Suspends the callingtask. The calling task is suspended until the identified process terminates or aborts or until a time limit is exceeded (see WAKEUP primitive of kernel); supported by the kernel.

INVOKE_PROCESS: Scheduling algorithms can now attempt to dispatch this process.

CREATE_JOB: Creates a new root process node and control returns to the calling process; supportedby the kernel.

APPEND_RESULTS: Appends results in the proper place in the return list; supported by the kernel.

WRITE_RESULTS: Replaces the valueof the results attribute with an item which is the value of the results parameter; supported by the kernel.

GET_RESULTS: Returns the value of the results; supported by the kernel.

STATUS_OF_PROCESS: Returns the current status of the process; supported by the kernel and enhanced by the STATISTICS package.

GET_PARAMETERS: Returns the value of the parameters; supported by the kernel.

ABORT_PROCESS: Aborts the process identified and forces any processes in the subtree rooted at the identified process to be aborted; supported by the kernel.

SUSPEND_PROCESS: Suspends the process. If this process is a parent of other process nodes then these other nodes are likewise suspended; supported by the kernel.

RESUME_PROCESS: Resumes the execution of a process. If it is the parent of other processes they are likewise resumed; supported by the kernel.

HANDLES_OPEN: Returns a number representing the number of node handles that the current process has open.

IO_UNITS: Returns a number representing the number of GET and PUT operations that have been performed by the process.

START_TIME: Returns the time when a process began execution; supported by the kernel and used by the STATISTICS package.

FINISH_TIME: Returns the time when a process completes; supported by the kernel and used by the STATISTICS package.

MACHINE_TIME: Returns the value of the amount of CPU time needed by this process; supported by the kernel and used by the STATISTICS package.

In summary, much of the basic support of the CAIS PROCESS_CONTROL package is performed by the kernel of WIS OS. The PEM needs to implement the procedures of this package to provide the high level program execution support necessary to run an Ada program. Further, the PEM needs to perform the actual scheduling of tasks which is not defined by the CAIS. The scheduling function is divided into two parts: a local scheduler package and a global scheduler package. These schedulers, in turn, require information about tasks, processes, hosts, and the network. They obtain some of this information from the STATISTICS package.

9.2.2 Local Scheduling

Local scheduling is handled by separating policy from mechanism. The kernel supports a priority based dispatcher as the basic mechanism for dispatching (see Section 3.0). This enables the kernel to provide an extremely efficient context switch when a new task is to be executed. The policy for local scheduling is implemented in the local scheduling package as part of the PEM.

The local scheduling policy must be implemented in a distinct module that can be easily varied according to the requirements of the processor and environment. For example, various local scheduling policies should be implemented for both uniprocessors and multiprocessors. In the uniprocessor case several of the following policies should be available: First-Come-First-Served (FCFS), shortest job first, a priority scheme which includes the ability to set a fixed high priority for certain tasks, round robin, multi-level feedback queues, and deadline scheduling support. Several multiprocessor scheduling algorithms should also be eveloped and implemented in Ada. Guidelines should be provided for the choice of scheduling algorithm with these guidelines being validated by simulation, mathematical analysis or literature references. It is also necessary to implement a package, STATISTICS, that collects statistics concerning the execution of programs (see Section 9.2.4). Note that the collection of status information concerning the execution of processes is stipulated by the CAIS. Appropriate extensions to handle distributed programs are required.

The local scheduler interface (parameters and functionality) is described next. In addition, the local scheduler needs to access the data structure that represents the set of ready tasks used by the kernel to dispatch.

LOCAL_SCHEDULER (PROCESS_NAME, TASK_NAME, POLICY, PRIORITY, DEADLINE, STATE_INFORMATION);

Functionality: A given Ada program runs as a process (PROCESS_NAME), and is describedby a CAIS process node. There will be at least one task per process (TASK_NAME) supported by the kernel task. The task is the dispatchable entity, but the LOCAL_SCHEDULER will use data rom the process node and ktask to schedule this entity. Ada semantics specifies that a task of a given Ada program has static priority with respect to the other Ada tasks of this program. The LOCAL_SCHEDULER must account for this when implementing the various policies. The solution is to have a base priority for the process. This base priority is subject to change based on the characteristics of the process and system. The priority of the tasks within the process can then simply be kept constant relative to each other.

The remainder of this interface lists optional parameterswhich are required only under certain circumstances. For example, a given site may be content with one policy, so there is no need to specify it. Further, it is not expected that at a given site that all policies will co-exist; in fact, it may not make sense for several to co- exist (e.g., FCFS and shortest job first), while it does make sense for others to co-exist (e.g., FCFS, a priority scheme and deadline support). The PRIORITY field would be specified as the initial or current priority, if priority scheduling were being used. The LOCAL_SCHEDULER might change the priority depending on its logic, but probably include certain priorities which are unalterable. The DEADLINE parameter would only be used for deadline scheduling and it would specify the task's deadline. The STATE INFORMATION parameter might be needed for certain policies, e.g., to specify the current state of the host. On the other hand, for efficiency, this information might be obtained directly from kernel data structures or the STATISTICS package.

In short, specific algorithms should be developed for each of the policies listed, feasible combinations of policies determined, and the LOCAL_SCHEDULER made efficient.

9.2.3 Global Resource Management

Each LAN should be highly integrated, sharing resources when possible or necessary. This sharing should reduce cost, and increase performance andreliability. However, we note that this part of the requested work is of a research nature. The discussion of global resource management with a list of objectives follows:

OBJECTIVES:

a. To develop a policy for sharing resources in the LAN that is consistent with WIS security requirements (B3) and the node model of CAIS.

b. To ensure that the global resource manager itself be fault tolerant.

c. To use the CAIS system model for representing resources, access to resources, and operations on the resources, and to identify extensions required to the CAIS to handle fault tolerance, distributed programs and the distributed environment.

d. To design and implement two or more global resource management schemes that are dynamic and flexible (global here with respect to the LAN).

e. To identify any problems with the Ada language for implementing the proposed algorithms (see [WELL 84]).

f. To provide specific solutions to common problems of distributed resource management such as deadlocks, instabilities of task movement, unfairness, etc.

g. To compare the performance of alternative solutions.

Research is required to develop an operating system structure and algorithms (based on the CAIS) that can perform global resource management in a cost-effective manner in a large, ever-changing, heterogeneous distributed system. The highly integrated sharing policy is to be with respect to the LAN, but the design should not preclude use of remote (long distance network) resources. The following scheduling issues need to be analyzed and solved:

a. The type and amount of state information needed to perform effectively.

b. What prediction techniques might be used to forecast future loads on the system.

c. How to deal with various timing issues (such as the delays in the subnet, the frequency and speed of the scheduling algorithm itself, etc.).

d. How to deal with the allocation of replicated objects.

e. What form of cooperation and synchronization should exist between distributed scheduling entities.

f. How to ensure stability.

For more discussion of these issues see [STAN 84]. Specific scheduling algorithms should be studied for applicability to the WIS environment, for example, those algorithms based on bidding, reverse bidding, or clustering of processes which communicate with each other frequently or in large volume or both [STAN 84] [STAN 85].

The global resource management algorithms should adhere to the "policy" and security constraints as required by WIS. Various forms of hierarchical and distributed control of resource management should be investigated. Advantages and disadvantages of various alternatives must be itemized. The resource manager must be fault tolerant and be able to handle uncertainty of the state of other hosts, missing or erroneous information, and the delays inherent in distributed systems. It must address issues such as the existence of unique resources, existence of trusted and untrusted sites, when and how to perform remote access versus moving the data or program, and whether to allow a task in execution to move. Strongly desirable is to have tasks capable of changing sites at any time during their execution. If the dynamic movement of tasks in execution is supported, the impact on the design should be identified.

Specific solutions are needed for problems such as deadlock, instabilities of task movement, unfairness, orphans, replicated objects, etc. Simulation can be used to compare relative performance of alternative algorithms. Two of the best algorithms as determined by the simulation should be implemented as Ada packages and evaluated on the testbed. By implementing these algorithms the feasibility of using the Ada language to implement such algorithms can be studied. For example, due to the highly dynamic nature of WIS it may be necessary to delay binding of logical objects to physical objects at execution time. The Ada language does not support this feature. Furthermore, in terms of IPC, the Ada language only supports the rendezvous. More flexible IPC mechanisms are necessary for decentralized resource management algorithms. The final report should include a discussion of the feasibility of using the Ada language to implement decentralized global resource sharing algorithms [WELL 84], as well as a discussion of the fault tolerant aspects of the resource manager itself.

The Ada packages to be delivered are GLOBAL_RESOURCE_MANAGER_1 and GLOBAL_RESOURCE_MANAGER_2 for the two algorithms chosen. The GLOBAL_RESOURCE_MANAGER_n packages interface to their distributed counterparts is via the IPC mechanism of the kernel. A more specific interface is not possible because it is highly dependent on the algorithm chosen. The GLOBAL_RESOURCE_MANAGER_n packages would also invoke the STATISTICS package to obtain information in making its decision. Since the GLOBAL_RESOURCE_MANAGER_n packages move jobs around the network, it needs interfaces to many different modules.

9.2.4 The Statistics Package

The STATISTICS package would be primarily used from the LOCAL_SCHEDULER and GLOBAL_RESOUCE_MANAGER_n modules. It should contain enough system data to aid these scheduling modules in making good decisions. For example, the STATISTICS package must be able to obtain information about processes from the CAIS node representation via the STATUS_OF_PROCESS function and others, as well as about system performance data. This may require extensions to the kernel to keep track of such data. The specific interface for this package is left to the contractor since it is a local interface, it is dependent on the scheduling algorithms implemented, and it will not affect portability.

9.3 A Note on Deadlock Resolution

At the task level, deadlock resolution should be handled by timeouts. This precludes the need for deadlock avoidance or expensive deadlock detection schemes. An alternative may be proposed; for additional information, see [HO 82], [OBER 82], [JAGA 82], [CHAN 82], [SINH 85] and [SHOU 85].

10.0 AUTHENTICATION SERVER

The authentication server provides authentication, key distribution and some aspect of naming. The security model is described first in brief.

10.1 Security Model

WIS must implement a multi-level secure system corresponding to at least the B (mandatory protection) level, ideally B3 with feasibility of going to A1 with the appropriate verification tools and work.

The system has a designated trusted computing base (TCB). One avenue in attempting to make the system verifiably secure is to make the TCB as small, well-defined and well-structured as possible.Clearly, the kernel is part of the TCB. It provides normal processes that execute at different security levels and ensures that there is not interaction between processes of different security levels. Actually, the kernel operations may also allow for "read-up", higher clearance levels reading from lower levels. However, write-up is not provided so as to simplify the integrity problems with the system. The kernel also supports reclassifier processes that may move data between security levels. In particular, reclassifiers are used to implement controlled "write-down". Each reclassifier process is also part of the TCB since misbehavior of a reclassifier constitutes a security violation.

Thus, the kernel enforces multi-level security for entities with known security levels. It provides also the creating of new processes with security levels inherited from their creators. However, it is the authentication server that assigns a security level to a new entity. Clearly, the authentication server is also part of the TCB. The authentication server performs several services, including authentication, key distribution and security logging.

10.2 Authentication

The authentication server maintains a collections of accounts which are createdby the security officer at a particular clearance level. A user must authenticate himself with the authentication server before making any non-trivial use of WIS. This entails communicating account name and password (or key) to the authentication server. The communication itself might be encrypted using the password as a key. The authentication server then decrypts the message and checks the correctness of the password. One rather inefficient but secure approach is for it to attempt to decrypt it with every account password and check correctness in this fashion. This would avoid sending the account name in the clear.

Assuming the authentication request was valid, the authentication server communicates with the kernel executing the requesting process and requests that the kernel change this process to the specified account and security level. This process relies on secure, unforgeable communication between the authentication server and each copy of the kernel. In particular, there must be absolute safeguards against an impostor authentication server. The authentication server logs all such authentication requests, whether successful or not.

10.3 Key Distribution and Encrypted Communication

The authentication maintains a secret secure key associated with each account. This is used primarily for secure communication with the authentication server. To communicate with other service modules securely, the authentication server provides "conversation keys". This is to minimize the use of the principal keys; that is, compromise of a conversation key is less critical than compromise of the principal account key.

In particular, a module A that wishes to communicate securely with another module B requests a conversation key from the authentication server. This is communicated back to A using A's principal key. The communication also includes an encrypted form of the conversation key, encrypted with B's private key. This allows B is ensure that A was authorized by the authentication server to engage in this communication, and that A is really A. Further details of this design can be found in the paper by Birrell [BIRR 83] on secure remote procedure calls.

10.4 Security Logging

The authentication server should provide a logging facility that records all security-sensitive events. These actions taken by itself such as each authentication request and its result. It also includes actions the kernel takes, like creation of reclassifier processes, messages from the authentications server and any attempts at security breaches.

10.5 Unresolved Issues

The basic authentication server is fairly simple. However, there are a few issues to resolve.

First, there needs to be a means of positive identification of the authentication server. One approach is to hardwire the address of the authentication server into software and have the network guarantee that impostors cannot generate this network address. This is adequate for the prototype but not sufficient in the final system.

Second, there are some design issues in the choice of encryption algorithm to use and efficiency implications. Currently, using a DES chip on each machine looks feasible. However, a simple XOR encryption seems adequate for the prototype (in software).

11.0 PIPES: SYMMETRIC INTER PROCESS COMMUNICATIONS

A pipe is a synchronized file that allows one or more readers and writers to transfer data, using the pipe as a bounded buffer. A familiar implementation of pipes is the UNIX operating system.

The pipe facility provides symmetric inter-task communication in the I/O model of communication. That is, the program or task that writes its output to a file can be connected by means of a pipe to the input of another program or task that reads from a file. A series of two or more tasks interconnected by pipes to run concurrently is commonly called a pipeline. Such a task pipeline executes in parallel in a similar manner to the pipeline structure used in a hardware processor. Each "stage" of the pipeline processes its input data and passing the resulting output to the next stage in parallel with the execution of the other pipeline stages.

The pipe facility complements the asymmetric inter-task communicationb provided by the Ada rendezvous mechanism. With the rendezvous mechanism, one task is a client who invokes entries; the other task must be a server who passively waits for an accept or select statement to complete. In plumbing terminology, the client has a male sex connection while the server has a female sex connector. In this analogy, a pipe provides a connector with two female ends, allowing two tasks with male ends (clients doing entries) to be plugged together. In addition, a pipe provides some amount of buffering to allow greater concurrency between the two communication tasks.

While there are other options for the design of such a "sex matching" inter-task facility, we choose to implement this facility in the file model, using the CAIS-specified "queue files" as the program interface for applications. A queue file in CAIS represents a sequence of information that is accessed in a first-in, first-out manner. There are three kinds of CAIS queue files: solo, copy and mimic. The solo queue file corresponds to the UNIX pipe. It is empty when created initially and then operates like a standard queue. That is, all writes append to the end of the file and all reads are destructive reads of the beginning of the file. A copy queue file operates like a solo queue file except that its initial contents are copied from another specified file. A mimic queue file is similar to the copy queue file except that writes to the mimic queue file are also appended to the file from which the queue file received its contents. See the CAIS Specification Manual [CAIS 85] for further details of the creation of queue files and operations on queue files.

Queue files shouldbe implemented by a concurrent Ada program that runs in one or more dedicated process nodes within each WIS cluster. In fact, there are performance advantages to having a queue file server or pipe server on every WIS node that uses queue files. In particular, it is more efficient for two tasks or programs running on the same machine to communicate by a queue file implemented on that same machine, rather than transmitting the data across the network. Furthermore, it is advantageous to have the queue file implementation on the same machine as at least one of the clients of the queue file, in the case where the clients (the reader and the writer) are running on different machines. This reduces the communication from two network transfers to one. However, remote clients can access a queue file implementation so having a local implementation of queue files can be viewed strictly as an optimization.

A simplified implementation of queue files or pipes is possible if we assume that the reader and the writer of the queue file is fixed at the time the queue file is created. Under this assumption, either the reader or the writer can include in its run-time support (within its address space), code for "eversing sex" to match the other clients plus some buffering. This implementation is rejected despite its performance advantages because it does not allow one to change the reader or writer and significantly complicates the run-time code in each client. Note that this does not preclude simple "bounded buffer" facilities for tasks sharing data in the queue file model within one address space.

All three types of queue files should be implemented by a concurrent Ada program structured much like the file server. (It could be included in the file server program although it is appealing to have this facility available without having the entire file system code resident.) In the simplest case, the queue file server consists of a simple task that provides entries for creating, deleting, opening, reading, writing, closing and querying queue files. The data in each queue file is simply stored in virtual memory. For greater potential parallelism, there could be one task per queue file, allowing the use of the select mechanism to synchronize readers and writers, similar to the example shown in the Ada LRM, Section 9.12[1815A 83].

Copy queue files simply require an initialization of contents from a given file and are otherwise implemented the same as solo queue files. Mimic queue files require that writes to the queue files also result in writes to the original file, another small extension on solo queue files.

This Ada program is expected to be fairly modest in size and might most reasonably be combined with the file system/storage management programming effort.

12.0 PRINTER SERVER

The printer service is basically just an output service. Data is written to the printer service, just like a file. However, the printer service also has a separate class of operations to query its state and modify its operation. In this vein, there are three aspects to the printer service: the data output, print job queuing query and modification, and printing parameter control and query.

12.1 Printer Output

The printer output is handled using standard Ada file writing with access to the printer file (actually a virtual printer which is then spooled for the printer) established using the same interface as specified for ordinary disk files, and hopefully similar to that in CAIS.

Thus, to output to a printer, the program opens a file using a printer service name. This open file is just a ordinary disk file created by the printer service. When the file is closed, the file is queued to be printed, and presumably deleted once printed.

12.2 Printer Queue Management

A particular printer service may have several printers plus print jobs arriving (at times) faster than they can be printed. Therefore, as standard, the print jobs are queued until a printer is available. A large printer service would have its own file system, presumably using standard file system software. Operations are required for querying the print queue, changing the priority of print jobs, deleting a job from a print queue, aborting a print job (in the middle of printing) and restarting or backing up a print job (to handle the case of the printer jamming, for instance).

12.3. Printing Control

A set of operations are provided to control how jobs are actually printed. These include operations for changing printer characteristics, changing fonts and paper, and other aspects of the printing.

13.0 MULTI-WINDOW DISPLAY SYSTEM

The multi-window display system (the display system) is a service module that manages a display (monitor), allowing multiple applications to share the display simultaneously.

The design of the display system is considered part of the overall operating system design for two reasons. First, the display system manages a device, the frame buffer and monitor, and device management is generally handled by the operating system. As a minimum, access to raw devices is provided and controlled by the operating system. Second, the display system manages the concurrent access to the display from multiple independently written applications. This coordination of different applications in sharing one resource (the screen in this case) is also part of operating systems design. However, there are clearly issues in the design of the exact primitives provided by the display system that are of domain of the graphics design team. For instance, the choice of GKS over PHIGS should be based on application requirements, graphics issues, and not the operating system design. To this end, one of the objectives in the design sketched out here is to provide a clear delineation of operating systems issues versus graphics issues in the display system design.

The basic approach is for the operating system design to provide an overall structure that allows the display system to "fit" well into the system with the other operating system services. This basic structure provides a framework on which a variety of different display facilities can be built. In fact, the ideal is for this structure to provide a configurable base for a variety of different display systems, much like the configurability that an operating system provides with device drivers. With device drivers, the operating system defines a basic set of operations that all device drivers should provide plus a base set of primitives that a device driver can use to implement its operations. The operating system does not define how to manipulate particular devices or the semantics of particular operations on a device. Similar, the display system should provide a framework for a variety of different display facilities as well as drivers for different hardware displays.

The overall design is described next, followed by an elaboration on particular aspects of the design, including display file types, window management, input handling, and a section comparing this design briefly to other approaches.

13.1 Design Overview

The display system provides one or more types of "display files". (This, unfortunately, has many meanings in the graphics community but hopefully our use of the term is reasonably consistent.) A display file provides the basic abstraction of an entity to which an application can write in order to display data on the screen. These can be thought of, and implemented as, open files. With certain types of display files, those providing the abstraction of display storage, it is also possible to read the display file. In this case, reading returns the data in the display file storage in the representation associated with that particular type of display file. For example, the frame buffer itself represents a fixed-size display file with display storage, namely the frame buffer memory. Writing to the file writes the frame buffer memory and reading the file reads the frame buffer memory. The "type" of display file determines the data representation it uses for display data, the ways in which it may be accessed and the particular operations provided on the display file. Associated with each display file is also a specification of its projection onto the display.

In general, the display system provides three classes of operations:

a. Operations for creating, destroying, reading and writing display files. The display system uses the standard Ada and CAIS I/O interfaces as much as possible here.

b. Modifying and querying the projection of the display file onto the screen. Theseare operations independent of the type of display file. Since these operations are not specified in the Ada or CAIS I/O interfaces, a possible set of operations is proposed in this document.

c. Querying the reverse mapping of screen coordinates to display files and addresses orcoordinates within the display files. There are no operations provided for these types of operations in the Ada or CAIS I/O interfaces as well. Again proposed is a possible set of operations in this document.

d. Finally, input handling is not strictly considered part of the display system module. Instead, a separate-input handler module interprets input and updates the display using the display system, the same as any other client of the display system. A subsequent section describes this approach in greater detail.

13.2 Display Files and Ada I/O

A display file is an open file object implemented by the display system that can be projected onto the display. A display file is written and read with the standard Ada interface for file writing and reading. That is, the display system implements one or more "external" display files. An Ada (open) file is associated with one of these external display files using the standard *CREATE* and *OPEN operations*, specifying a string name for the desired external display file.

A portion of the file name space should be reserved for the display system. For instance, "[DISPLAY.LOCAL.]<DISPLAY_FILE_NAME>" could refer to an external file on the local display system. "[DISPLAY.<HOST_NAME>]<DISPLAY_FILE_NAME>" would refer to the external file on the display system running on the specified host. Also that different prefixes to the <DISPLAY_FILE_NAME> be used to indicate the type of the display file. For instance, "[DISPLAY.LOCAL.]TEXT.VT100.FOO" could specify a particular display file that simulates a VT100-compatible text display file. The CAIS CREATE operation can use the FORM and ATTRIBUTE fields to specify parameters and particular attributes of the desired display file. The ATTRIBUTE field should be able to specify the "size" or dimensions of the display file. (This use seems consistent with the CAIS use of these fields, although the standard provides no guidance on display files, in particular.) For instance, "(TEXTLINES => 12)" specifies that the display file should provide space to display 12 lines of text. (The CAIS assumes that terminals are "physical" rather than "virtual" so it only provides for querying the attributes of a display, not specifying them.)

Other operations might be provided for defining new external files besides CREATE. In particular, the CAIS CREATE_NODE operation could be used to define new display file nodes representing a new class of display files. The OPEN operation can provide standard file access to existing external display files.

Following the Ada LRM, there are two forms of access to display files: direct access and sequential access. In direct access, the display file is viewed as a storage container for display data. Direct I/O writing to the display file updates this display storage at the current "index" for the open file. Direct I/O reading returns the display data at the current index. The representation of the display data is particular to the specific type of display file.

With sequential access, an open file with write access provides stream access to the display file. Data written to the stream is interpreted in a fashion specific to the type of display file. For example, for the TEXT.VT100 display file, sequential writing may provide a byte stream connection that is interpreted compatible with the VT100. Similarly, an open file with read access reads a stream from the display file whose form and semantics are specific to the type of display file.

An open issue is whether to require all display files to provide both a direct I/O and a stream interface. Requiring a direct I/O interface requires that every display file have some type of display storage that it implements. In conventional framebuffer architectures, this does seem like a problem, either at the hardware level or in the software implementation. It is the "dumb terminals" and exotic graphics hardware that will provide most of the problems. Since stream access can simply be a restricted version of the direct I/O, it appears easy to provide with all display files.

Further discussion of Ada and CAIS support is provided in the context of display file types.

13.3 Display File Types

There are several types of display files, according to the representation of data used in the display file and the operations on the display file data. Three basic types of display files are:

a. Pixel (or bitmap) display files

b. Text display files

c. Graphics display files

The only type of display file covered by the Ada LRM or by the CAIS is the text display file. The text display file is compatible with the Ada and CAIS specifications. In addition, a fully implemented display system should provide display file types corresponding to the CAIS extensions for scroll, form and page terminals. The Graphics Task Force should and will want to further define the specifics of the representations, especially for the structured graphics display files.

13.3.1 Pixel Display File

The pixel or bitmap display files provide a model of display in which data is represented as an array of pixels of some dimensions. There are two basic forms one might provide. First, a basic device-dependent display file corresponds to the raw framebuffer plus one can provide display files with the same pixel depth (i.e. 1 bit pixels, 8 bit pixels, etc.) and resolution plus pixel size as the physical frame buffer but different X and Y dimensions (both larger and smaller) than the physical framebuffer. The second type is to provide a virtual pixel display file which is device independent and mapped to the actual display as well as possible. This would allow an application using pixels to be reasonably device independent, an important issue in a distributed heterogeneous environment.

A pixel display file is implemented as a "random access" file with the byte-offset in the file indicating the X and Y coordinates in the pixels. For instance, a 32-bit file byte offset position could be interpreted as two 16-bit coordinates in the pixel file.

In the main mode of use intended for pixel display files, display data is generated by the application and then written to the display file provided by the display system. In the case of interactive graphics editing-style applications, a change in the data is performed on the application copy and then this copy is transmitted to the display system.

Proposed is that the stream access to a pixel file be interpreted as stream so-called rasterop or bitblt operations. For example,

> CLEAR (DF: in DISPLAY_FILE;): Clears the specified display file. This would be implemented by writing the command code for "Clear" to a sequential file associated with the specified display file.
>
> PIXEL_COPY (DF, FUNC, SRC_X, SRC_Y, DST_X, DST_Y, WIDTH, HEIGHT:); Copies the pixels from one region to another using the supplied pixel copy function (i.e. AND, XOR, Copy, etc.) It would be implemented by writing the encoding for this operation and its parameters to the writeable stream attached to the display file. In general, a complete set of operations might be modeled after the X raster graphics primitives, or some extension thereof, e.g., bitwise AND, invert, bitwise OR, PixelFill, Draw, etc.

These operations allow the application to modify the data in the display file with substantially less communication cost that performing the operations locally and then transmitting the changes. They also allow the display system to provide efficient access to and use of graphics support hardware for (for example) so-called rasterop or bitblt operations. For example, an XOR pixel copy provided by the display system may be implemented by the graphics hardware in the framebuffer. (Note: the semantics are considerably simpler than the X semantics because the display file stores all the pixels, not just the ones being displayed.)

The pixel display files support applications that wish to deal directly with pixels.

13.3.2 Text Display Files

A text display file is one in which the mapping from bytes to pixels is defined in terms of a font. (defined as a set of similar-sized pixel patterns). For example, the bytes may be interpreted as ASCII and the pixel representation realized by mapping each byte to a Time Roman ASCII font. The font to use is one of the attributes that may be specified in theCREATE operation. Using text display files, the application can deal with character strings and not be concerned with mapping to graphics or pixel images. The display system provides translation of bytes to pixel images according to a specified font and additional information, such as the handling of line wraparound. Operations are provided for loading, querying, and deleting fonts. Again, the X window system can be taken as a general model.

One type of text display file required is one that emulates a standard type of terminal, such as a VT100. In addition, display files should be provided that implement the scroll, page, and form terminals specified in CAIS. In general, this support seems best provided using the stream access to the text display files, with interpretation of the stream data to implement the desired operations. In this vein, the read stream provides a path for return values to query operations and such like sent on the write stream. Similarly, the read stream provides a return path for updated forms in the implementation of the CAIS form terminal.

The details of text display files seem adequately covered by Ada and CAIS specifications. [1815A 83] [CAIS 85] An implementation should provide display files corresponding to these specifications as well as a display file that emulates a standard ASCII terminal on its output stream connection.

Each such open file defines a window. Windows are mapped to the screen into one or more viewports. In this vein, a window is a world coordinate system for output (particularly relevant for graphics), where as a viewport is a portion of the window that is mapped to the screen. In particular, changing the attributes of a viewport or viewports is not apparent to the application.

13.3.3 Structured Graphics Display Files

Structured display files provide high-level display data representation both in their display storage as well as on data sent on their stream access connections. For example, the read and write streams of a structured graphics display file might provide interpretation that supports the CORE standard, GKS or PHIGS. Used with direct access I/O, the elements read and written represent high-level representations of graphics objects.

In general, the display system should allow the implementation of a wide variety of different structured display files. Referring back to the device driver analogy, it should be possible to configure the display system with different combinations of display file types, depending on the local application requirements and display devices available.

Further specification of the structured graphics display files is deferred pending discussions with the Graphics Task Force.

13.4 Display File Projection Operations

For display file data to be visible, it must be projected onto the display screen. (There may be several display screens in some cases.) An application can write and read a display file independent of whether the display file is visible or not. The "projection control" associated with both windows and viewports is discussed. Also, noted is that neither the CAIS nor the Ada I/O interfaces specify any aspects of projection management because neither standard deals with virtual terminal or multi-window display systems. However, various graphics standards do provide operations that may be suitable. The following provides a basic set of operations for discussion. These may well be replaced by some standard set recommended by the graphics task force. However, these operations are proposed as the OS interface to the display system. Standard Ada packages might be layered on top of these operations to provide GKS, CORE, etc., standard operations.

The basic operations provide for creating, deleting, querying and modifying projections. A projection provides a coordinate system plus specifies projection from the display file to this coordinate system, plus projection from this coordinate system to the screen. In addition, one can group projections, possibly from separate display files, and manipulate a group of projections as a single entity. The specific operations are described below.

CREATE_PROJECTION (PROJID, DISPLAY_FILE, INITIAL_SPEC): Creates a new projection associated with the specified display file, returning an identifier, PROJID, for the projection. The projection state include the screen space for the projection, the world coordinates for the projection plus the correspondence between the two coordinate systems.

QUERY_PROJECTION (PROJID, RETURN_RECORD): Returns the information about the specified projection in the RETURN_RECORD field.

MODIFY_PROJECTION (PROJID, MODIFICATION): Modifies the projection as specified. (There might be two forms of this - one to modify individual parameters as well as one to revise all the parameters.)

DELETE_PROJECTION (PROJID): Deletes the specified projection.

CREATE_PROJECTION_GROUP (GROUP_PROJID, INITIAL_PROJID): Creates a group of projections with initial member INITIAL_PROJID and return the projection identifier in GROUP_PROJID. A projection group can be modified, queried and deleted the same as for an individual projection.

JOIN_PROJECTION_GROUP (GROUP_PROJID, PROJID): Adds the specified projection to the specified group.

LEAVE_PROJECTION_GROUP (GROUP_PROJID, PROJID): Removes the specified projection from the specified group. A projection group ceases to exist when the last member leaves, as well as by calling DELETE_PROJECTION on the GROUP_PROJID.

The projection group mechanism allows groups of display file projections associated with a single application to be manipulated as a group. For instance, using a VLSI layout editor, all the "views" of the layout may be pushed to the background or brought to the top in a single operation, rather than individually.

Finally, many of these operations will be invoked only indirectly by the user through a user interface facility, providing screen management and "personality". Separating the basic services from the user interface design allows the display system to support a variety of different user interfaces with minimal code in the display system itself that is tie to particular "religious" preferences that abound in user interface design. A similar argument applies to input handling and is described on the following section.

13.5 Reverse Projection Support

The display system provides operations for mapping from screen coordinates back to particular display files and to objects or coordinates within the display file.

REVERSE_MAP_TO_PROJECTION (XCOORD, YCOORD, ZCOORD, PROJID, DISPLAY_FILE): Takes the specified x, y and z coordinates for screen space (z-coordinate is not always used) and return the projection and display file associated with these screen coordinates.

REVERSE_MAP_TO_OBJECT (PROJID, XCOORD, YCOORD, ZCOORD, OBJECTID): Selects the object in the display file associated with the specified screen coordinates, assuming these coordinates specify an area covered by this projection.

REVERSE_MAP_TO_LOCAL_COORD (PROJID, XCOORD, YCOORD, ZCOORD, LOCX, LOCY, LOCZ): Maps the specified screen coordinates to local coordinates within the specified projection, assuming they are so contained. Otherwise, an error is returned.

These operations provide basic a hit detection facility in the display system. Typically, the input demultiplexor module invokes REVERSE_MAP_TO_PROJECTION to determine which projection and display file, and therefore which application needs to receive the input event, if any. Then, it may pass on the coordinates as such or else map to local coordinates or an object-id, as is appropriate for that application. As an optimization, the input handler would "guess" that the last projection used is the projection in which the coordinates fall again, and then use either REVERSE_MAP_TO_LOCAL_COORD or REVERSE_MAP_TO_OBJECT with this projection. If those fail, the input handler would resort to explicit determination of the right projection and display file if those fail.

Besides these operations, reverse mapping operations may be provided specific to each display file, mapping local coordinates to specific display data within the display file.

13.6 Input Handling

The display system does not handle any aspect of input. It is purely a service for input interpreters and other applications. A separate input handler module should be developed that handles a common set of functionality, useful for a wide class of applications. In particular, it should provide keyboard character echoing and line editing with suitable options for disabling these interpretations. It should also provide tracking of pointing devices, such as a mouse, using cursor update in the display system to indicate changes in the mouse position. The specification of more exotic input, such as valuators and stroke input is left to the Graphics Task Force.

Finally, the display system must provide support for cursors and menus. Both of these are viewed as simply display files that are created and manipulated by the standard routines described in the section.

13.7 Comparison with Other Approaches

A display system presents an image of some portion of internally stored data or objects The display system is charged with providing a mapping from the data to the display and vice versa, for selection based on screen coordinates, such as used for hit detection It is also chargedwith maintaining consistency between the stored data and the display Thus, if some aspect of the display mapping is changed, it may have to regenerate the image from the objects. Similarly, if an object is changed, the display software should be informed and the display image updated accordingly.

There are three basic approaches in displaying the data

a. The data is logically stored in the application. When the application changes the data, it retransmits the data to the display system When the display system changes the display, it signals the application to retransmit the data to the display This approach is used by the SMI window system as well as the X window system (runs on the MicroVAX)

b. The data is logically stored in the display system When the application needs to change the data, it requests the changes in the display system which make the

c. Assuming a shared address space between application and the display system, the stored data can be shared and redisplay is invokedby whichever component caused the change. Cedar windows and SmallTalk use this approach.

changes and updates the display. When the display system changes the display, it redraws the display from its stored data. The Virtual Graphics Terminal Server (VGTS) done for the V distributed system uses this approach.

The VGTS approach, in a somewhat modified form was chosen. The Cedar/Smalltalk approach requires that all applications share the same address space. That is too restrictive and impossible if to allow remotely running applications to access the local display.

The first approach requires that all applications be structured to handle redisplay. This seems like a dangerous complication to impose on application programmers. Even if buried in a standard library, there are problems with synchronizing with application updates to the data being displayed.

13.8 Implementation Ideas

The display system should be implemented as an Ada program that provides entry procedures for the display functions described above. These entry procedures are typically invoked by the Ada remote procedure call facility provided by the basic WIS system. The display system would likely be three tasks (in a one monitor configuration).

a. Main task: Implements the rendezvous with the client applications.

b. Timer task: Performs periodic "housecleaning" jobs, such as blinking the cursor, if desired. Screen update may also be deferred using a timeout provided by the timer task.

c. Redraw task: Updates the screen according to changes in the display data structures. This could be part of the main task but is made concurrent purely to allow for parallel execution of the two tasks in future parallel implementation. Redrawing does tend to take considerable processing time, dependent on the complexity of the display and the hardware support provided.

Clearly, the redraw task and the main task need to be properly synchronized.

The kernel-provided interface to the display hardware depends on the general nature of the display hardware. In the simple and most common case, the hardware interface is an area of memory that corresponds to the frame buffer memory. In this case, the kernel allows the display system to map this memory into its address space, providing it with direct read and write access to this memory.

This general memory interface seems preferred. For hardware that does not fit this memory model, the kernel provides an "open file" interface, allowing the display system to read and write the raw device via this open file connection.

13.9 Summary

The display system provides an operating system module framework for implementing and configuring a variety of display files, corresponding to different graphics and text interfaces. The resource sharing and multiplexing aspects of the display system have been separated from the user interface and graphics aspects. In this vein, the input handling is also separated from the display system. A brief description of a basic input handling module is included. However, the full specification of input handling and interpretation is left to the graphics specifications and to the specification of the command interpreter/user interface.

14.0 COMMAND LANGUAGE INTERFACES

The command language interface is an important interface between various types of users and system facilities/capabilities. The command language interface will be human-engineered for both the casual and sophisticated user, as well as for users at different security levels. It will also be engineered to allow users to progress from casual, to more sophisticated, to highly sophisticated use of the system as they gain experience. A simple, easy-to-learn, subset of commands will be available for the casual user. The casual user will be working with fixed menus and icons. A more sophisticated user will have a command-line input facility along with the menu based facilities. Finally, the most sophisticated user will have the characteristics of a high-level programming language available and be able to create general scripts and macros as well as specific profiles to be automatically used in certain instances such as at login or at the invocation of an editor.

The programming-like interface may even be similar to the Ada language itself, but will, at a minimum, allow the use of some Ada statements, the invocation of package calls, and the creation and execution of libraries of Ada packages. For example, the programming-like interface includes conditional and loop control structures, argument handling, variables, string manipulation, named procedures, an some I/O instructions.

A simple form/menu/icon driven interface with appropriate prompting is to be supplied for terminals and simple workstations. However, menus are supported for various powerful workstations, including powerful multi-window high-resolution graphics workstations. Proper default values (although easily modified) will simplify the interface. The command interface will also support pipelining, including multiple pipelines and redirection. Pipelining can be defined as the standard output of one program acting as the standard input to another with the whole process to be run in a sequence. Redirection is the ability to easily change the destination for the output of the program. It should be possible to choose the destination both at command initiation time as well as for some commands during or even after the execution of the command.

The command language is to be well designed so as to provide a consistent syntax, as well as be common across diverse hardware systems. It will support interactive and batch jobs. Help and tutorial features will also be available. Various types of access control is supported at the command language level, including limited individual and group access by securitylevel and discretionary need to know limits. Users are able to create and manipulate data objects, invoke programs, and interface to external systems.

15.0 I/O DRIVERS

Enumerated below are the requirements for WIS I/O device drivers.

15.1 Common Interface to All I/O Devices

There will be a common interface to all I/O devices, with the device "name" or "address" as part of the call. The use of parameters and their sequences shall be as uniform as possible among the various devices and drivers. The device name will be mapped to a real device in the common interface module. A device "address" may be symbolic and may be similarly mapped. In general, this mapping will be transparent, and the user need not know much (if anything) about the device to which the mapping is made. Further, the mapping may be used to redirect the user's I/O transparently.

15.2 Driver Dependence

Each driver should (to the maximum extent possible) depend only on the device and not on the host system.

15.3 Driver Requirements

Drivers shall be written for all devices to be used anywhere in WIS, and to work with each machine used in WIS which connects to any of these devices. To the greatest extent possible, code for the drivers shall be common between devices; different code sequences should be written only when required.

Drivers should be written initially for those devices currently in use or projected to be in use in WIS. A specific list will be provided at a later date. The list of devices likely to be used somewhere within WIS includes:

a. Disk Drives: IBM 3330, 3340, 3340, 3350, 3380 disks; all common varieties of 8", 5" and 3.5" floppy disks; commonly available small hard disks.

b. Tape Drives: tape drives at 1600 bpi and 6250 bpi for all widely used computer systems (IBM, DEC, Honeywell, Sperry, etc.), and new IBM Tape Casette Drives.

c. Unit Record Equipment: card punch, card reader.

d. Communications lines: at all "usual" speeds (300, 1200, 2400, 4800, 9600, 19200, etc.), Ethernet interface.

e. Terminals: standard ASCII terminals, bitmapped high resolution terminals, graphics displays.

f. Printers: microfiche printer, line printers, laser printers (from small Canon engine up to large IBM and Xerox units), IBM 3270 type terminals, etc.

g. Miscellaneous: optical scanners, radar units, etc.

15.4 Signals

For performance reasons, drivers shall have the provision to return two completion signals: (1) after command is received and accepted, and (2) after I/O operation is complete. One or both signals may be requested at call time. Modules may be configured to provide one or both automatically, rather than relying on a call time specification, if that system has uniform requirements.

15.5 Read Checks

Storage device drivers shall have the ability to provide "read check", i.e., to write and then automatically read back and verify, prior to signalling completion. This read check shall be optional and shall be indicated as a parameter in the call.

15.6 Reliability and Fault Redundancy Features

Each driver shall have reliability and fault redundancy features. Specifically, where appropriate, automatic retry (N times for some appropriate N) shall be provided. There shall be automatic error logging. Associated with each driver shall be a diagnostic module which shall attempt to diagnose errors, and shall provide notification when repair or alternate device use is needed. Storage devices (especially disks) shall to some extent be self redundant, i.e., have alternate sectors for bad sectors, etc. I/O drivers shall be prepared to support this feature (if in hardware) or provide it in software.

15.7 Queuing I/O Requests

The I/O system shall be prepared to accept and queue I/O requests and issue them in sequence (if relevant) or in a manner that maximizes performance (FCFS, SSTF, etc.) Appropriate scheduling algorithms should be provided for each device, for disk, SSTF, SCAN, FCFS. Choice of algorithm should be justified either by simulation, mathematical models or literature search. Multiple algorithms shall be provided when different algorithms are required in different circumstances.

15.8 I/O Driver Configuration

It shall be possible to configure I/O drivers to omit those features not needed in a given installation or given circumstances, so as to save memory space and execution time.

15.9 Mandatory Ada Requirements for Drivers

Drivers shall be coded to the maximum extent possible using the Ada programming language. Where for performance reasons, lower level (assembler) code is required, provision shouldbemade to replace that code with Ada code when the compiler improves. Justification shall be provided for any use of assembler or machine code, other than use of the Ada programming language; such use of lower level languages is discouraged.

15.10 Device Substitution

The I/O system shall provide a mechanism whereby one device may be substituted for by another, provided that the functionality is equivalent, in a manner invisible to higher levels (except perhaps for performance). Such substitution shall even be possible across a LAN or a wide area network.

15.11 Standardized Interfaces for Drivers

To the maximum extent possible, interfaces for similar devices should be similar and standardized. For example, a "standard terminal interface" should be the front end to all terminals. All text-type terminals should support the same minimum set of commands, with more advanced terminals supporting additional features, defined as optional in the interface. Likewise, the disk interface should at the highest level provide a standard set of commands. At a still higher level, it should be possible to treat most or all devices as a serial bit or byte stream. Justification should be provided in each case as to why interfaces between devices of the same "type" or of different types are not the same. (And where there are layers, with commonality above a certain point and differences below it, the location of that point should be justified.)

16.0 HARDWARE BASE

The WIS OS is designed to be highly portable over a class of machines and networks. This section characterizes this hardware base plus describes the anticipated evolution of this hardware base. The latter is important to recognize so that the WIS OS software can be designed to accomodate and take advantage of new hardware. In general, higher priority is given to making the WIS OS portable over future machines and networks than having the design compromised to deal with the limitations of old hardware.

The hardware base is divided into several categories, namely communication, processing, storage, time and miscellaneous peripherals. Overall, the hardware base consists of local clusters interconnected by one or more wide-area networks or internetworks. Each cluster consists of a variety of nodes connected by one or more local networks. The nodes include medium to high performance personal workstations, file servers, database servers, printer servers, computation servers and gateways. Further details are described in the following sections.

16.1 Communication

The local network technology provides one megabit or greater data rates between any two nodes in a cluster. Thus, for example, it is feasible to provide file service and program loading to workstations from a (shared) cluster file server. The local network also provides multicast datagram service to subsets of the local nodes, such as is provided in the Ethernet. The local network is assumed to provide low delay and low error rates, as is typical of local networks.

In the final hardware version, requirements may be added for robustness under failure conditions. For example, most local networks fail in total with any break in the cable. Similarly, a station or transceiver failure can also disable the entire local network. It seems appropriate to require that a cable break only disrupt communication across that portion of the cable. Also, a station failure should only disrupt communication with the associated host. Finally, there needs to be some provision for local network failure in which packets are flooding the network at a rapid rate, overwhelming hosts with the processing load of receiving, processing and discarding the packets. As a failure mode, it may be handled by using multiple local networks and having host network interface software that disconnects from the network when bombarded with packets. As a security problem, there may need to be phsyical security precautions taken with the network.

The final version local network must be provided with reasonable physical security, i.e., protection against jamming, breaks and unauthorized taps. Communication above a certain security level is expected to be encrypted on the local network while below that level to be sent in the clear for efficiency reasons. The local network will be secure at some level.

Local networks are connected to wide area facilities by gateway nodes. A gateway node is expected to be a full computer of substantial processing and memory capacity. The gateway has more duties than a standard "packet shuffling" gateway commonly found in datagram-based internetworks. In particular, the gateway serves three functions: translator, isolator, and authenticator. These functions are detailed later in this section. However, the key point is that the gateway is expected to perform functions well beyond most current gateways and have the commensurate hardware facilities.

The wide area networks are provided by separate agencies. The data rates must be expected to be lower, the delays longer and (most importantly) the predictability of service and service delays much poorer. The following are some proposed requirements for the wide area networks.

a. First, they should provide as high a data rate as possible under normal operating conditions. It seems entirely feasible to provide megabit rates using satellite and long distance fiber optics links.

b. Second, the network should provide multicast capabilities whenever feasible. The gateways should provide a basic multicast capability, similar to the local network facility, and any such facilities in the wide area network could be used to improve the efficiency of this multicast facility.

c. Third, the wide area networks should provide graceful degradation under failure or attack. This means two things. It is better for the data rates between nodes to degrade under failure than fail altogether. Total failure should never occur. Also, the wide area network should be able to provide a cluster with an indication of its operating level in terms of data rate and delays. This allows local clusters to use different strategies, depending on the operating conditions of the wide area networks.

Finally, the wide area networks should be as physically secure as possible. However, it is not clear what degree local clusters can rely on wide area networks for secure communication in terms of encryption. Local clusters will have to encrypt communication before it leaves the cluster. Over time, the data rates for both local and wide area networks will improve.

16 2 Processing

Processing power in a cluster and certainly across all of WIS is distributed, heterogeneous and substantial. Each processor has at least 32-bits of addressability. Each node has at least one 1-MIP (a million instructions per second)processor and a substantial amount of memory. Provision must be made for different instruction sets, different byte and word ordering as well as different word sizes and data representation (as in floating point representations). Nodes may be uniprocessor or multi-processor. Multi-processor nodes, should be configured as n identical CPU's connected through local caches and a memory board to common memory. Processors without such a common view of memory are viewed as special purpose processors, similar to device interfaces. For instance, a processor located as a network interface board serves to augment the functionality of the network interface but is not considered one of the CPU's, even if it is the same microprocessor

There are three "flavors" of processing in a cluster:

a Personal processing

b Service processing

c Computation processing

Personal processing is largely interactive user interface processing, including text editing, graphics, spreadsheets, mail, etc. This processing is primarily provided by personal workstations, one per user. Personal workstations will provide memory protection and mapping in addition to the characteristics described above. WIS may contain a significant number of multiprocessor workstations over time, with the multiple processors providing better response, more total compute power and a higher degree of reliability, i.e., ability to continue after a processor failure.

Service processing refer to the cycles required by file servers, gateways, print servers and the like. Services will be largely run on dedicated network nodes, i.e., dedicated shared database machines rather than distributing the database across workstations. The service machines will be built with similar hardware in most cases to the personal workstations. That is, the file server should use the same processor, backplane, etc., as the personal workstations if possible. This simplifies software and hardware maintenance as well as personnel training. The additional capacity of server machines can be provided by adding additional processors, memory and special peripherals. Additional service capacity can also be achieved using multiple server machines.

Finally, computation processing refers to executing very large compute-bound jobs. This capacity is provided (when needed) by special "computation servers" on the network. To allow considerable flexibility in the choice of such machines and to minimize the overhead on their operation, these machines are treated as essentially special peripherals. A program is sent to a computation server to be executed as input and the results are returned as output. Provision is made for interacting with the job during execution. However, the program in execution has limited access to facilities outside of the computation server. In particular, it does not see the transparent distributed program environment that is otherwise provided in WIS. For example, a WIS cluster that needed to do a sophisticated weather project might acquire a Cray or equivalent machine. A simple interface program would be written for the Cray or its network interface processor to "speak" the WIS protocols. The standard Cray execution environment would remain unchanged.

In general, this approach may be taken in bringing in non-standard hardware and software. The facility is interfaced to WIS using a simple "facade" but not integrated fully into the WIS environment.

16.3 Storage

Most storage in a cluster is provided by standard, high capacity Winchester technology disks. In general, there are hundreds, if not thousands, of megabytes of storage available in each cluster.

Floppy disks might be provided for personal use. However, there is some motivation to ban them altogether as a security problem. They provide a high data rate, inconspicuous data channel not in human-readable form, out of WIS.

Significant use of optical disks is expected in WIS over time for several purposes. For one, optical disks can be used for backup of on-line storage, replacing tapes. Optical disks will also evolve as a distribution mechanism for large amounts of data.

16.4 Time

TBD

16.5 User Interfaces

TBD

16.6 Miscellaneous Peripherals

TBD

16.7 Prototype Hardware

TBD

17.0 TIME SYNCHRONIZATION AGENT

The notion of which event "happened before" another one is usually based on physical time. In a distributed system this notion would implicitly assume that some universal time can be defined and observed identically from various locations. This is impossible. Practically, one may approximate some universal time with a given accuracy, and use this approximated universal time to develop a relative ordering of events, either a partial ordering or a total ordering---whatever is required.

Consider that a logical clock can be described as a function F which assigns a number to any action initiated locally. Such logical clocks can be implemented by a simple counter. Now consider a distributed system where each producer process owns a logical clock. The problem then is to guarantee that the system of clocks satifies some condition Z so that a particular ordering may be built on the set of actions initiated by the producers. In general the ordering is not unique and may not be equivalent to a chronological ordering. The objective of the Time Synchronization Agent is to obtain a unique physical time frame within the system so that consistent schedules may be derived from a total chronological ordering of actions occuring in the system.

When multiple physical clocks are involved (as is true in a distributed system) and to form a total order, it is not enough that the clocks run at approximately the same rate. They must be kept synchronized. It is possible to synchronize the local clocks of various processors in a distributed environment with an accuracy that is limited roughly by the sum of errors which accumulate because of different clock rates in each computer, and errors arising from the uncertainty about the time for communication between machines. The clocks must be synchronized so that the relative drifting of any two clocks is kept smaller that a predictable amount. In [LAMP 78] a solution to accomplish this is given. This agent should implement this solution or one similar to it. This agent should also implement a second solution that is based on having all the clocks synchronize with some globally available time signal.

Distribution List for IDA Paper P-1893

Sponsor

Maj. Terry Courtwright WIS Joint Program Management Office 7798 Old Springfield Road McLean, VA 22102	5 copies
Maj. Sue Swift Room 3E187 The Pentagon Washington, D.C. 20301-3040	5 copies

Other

Col. Joe Greene STARS Joint Program Office 1211 Fern St., Room C107 Arlington, VA 22202	1 copy
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2 copies

CSED Review Panel

Dr. Dan Alpert, Director Center for Advanced Study University of Illinois 912 W. Illinois Street Urbana, Illinois 61801	1 copy
Dr. Barry W. Boehm TRW Defense Systems Group MS 2-2304 One Space Park Redondo Beach, CA 90278	1 copy
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1 copy
Dr. Larry E. Druffel Software Engineering Institute Shadyside Place 580 South Aiken Ave. Pittsburgh, PA 15231	1 copy

Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1 copy
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1 copy
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1 copy
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1 copy

IDA

General W.Y. Smith, HQ	1 copy
Mr. Seymour Deitchman, HQ	1 copy
Mr. Robin Pirie, HQ	1 copy
Ms. Karen H. Weber, HQ	1 copy
Dr. Jack Kramer, CSED	1 copy
Dr. Robert I. Winner, CSED	1 copy
Dr. John Salasin, CSED	1 copy
Mr. Mike Bloom, CSED	1 copy
Ms. Deborah Heystek, CSED	1 copy
Mr. Michael Kappel, CSED	1 copy
Mr. Clyde Roby, CSED	1 copy
Mr. Bill Brykczynski, CSED	1 copy
Ms. Katydean Price, CSED	2 copies
IDA Control & Distribution Vault	3 copies

END

4-87

DTIC