

PROJECT ADMINISTRATION DATA SHEET

ORIGINAL REVISION NO. _____

Project No. G-36-612 DRS. RICHARD LEBLANC + PARITHA DASGUPTA GTRI/XXX DATE 11/30/83

Project Director: DR. MARTIN S. MCKENDRY School/Dept ICS

Sponsor: NASA - Langley Research Center
Hampton, VA 23665 12-8-88

Type Agreement: Grant No. NAG-1-430 12-8-86

Award Period: From 1/9/83 11/8/85 (Performance) 12-8-86 (Reports)

Sponsor Amount: This Change Total to Date
Estimated: \$ _____ \$ 47,402
Funded: \$ _____ \$ 47,402

Cost Sharing Amount: \$ _____ Cost Sharing No: _____

Title: "A Support Architecture for Reliable Distributed Computing Systems"

ADMINISTRATIVE DATA

1) Sponsor Technical Contact:
Dr. Edwin C. Foundriat
NASA - Langley Research Center
FLT m/s 138 MS #772 469
Hampton, VA 23665
(804) 865-2077
3535

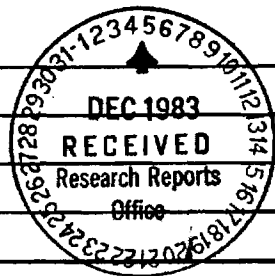
OCA Contact John W. Burdette ext. 4820
2) Sponsor Admin/Contractual Matters:
Mrs. A.S. Reed (MS 126)
National Aeronautics and Space Admin.
Langley Research Center
Hampton, VA 23665
(804) 865-3215

Defense Priority Rating: N/A Military Security Classification: N/A
(or) Company/Industrial Proprietary: N/A

RESTRICTIONS

See Attached NASA Supplemental Information Sheet for Additional Requirements.
Travel: Foreign travel must have prior approval - Contact OCA in each case. Domestic travel requires sponsor approval where total will exceed greater of \$500 or 125% of approved proposal budget category.
Equipment: Title vests with GIT; however, Government reserves the right to acquire transfer to itself title to items of \$1,000 or more.

COMMENTS:



COPIES TO:

Project Director
Research Administrative Network
Research Property Management

Procurement/EES Supply Services
Research Security Services
Reports Coordinator (OCA)

GTRI
Library
Project File

**GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION**

NOTICE OF PROJECT CLOSEOUT

Closeout Notice Date 06/26/91

Project No. C-36-612 _____ Center No. R5702-0A0 _____

Project Director LEBLANC R J JR _____ School/Lab COMPUTING _____

Sponsor RANDOLPH RESEARCH CTR, VA _____

Contract/Grant No. NAG-1-430 _____ Contract Entity BTRC

Prime Contract No. _____

Title A SUPPORT ARCHITECTURE FOR RELIABLE DISTRIBUTED COMPUTING _____

Effective Completion Date 900410 (Performance) 900410 (Reports)

Closeout Actions Required:	Y/N	Date Submitted
Final Invoice or Copy of Final Invoice	Y	910614
Final Report of Inventions and/or Subcontracts	Y	910614
Government Property Inventory & Related Certificate	Y	900119
Classified Material Certificate	N	_____
Release and Assignment	Y	_____
Other	N	_____

Comments _____

Subproject Under Main Project No. _____

Continues Project No. _____

Distribution Required:

Project Director	Y
Administrative Network Representative	Y
BTRI Accounting/Grants and Contracts	Y
Procurement/Supply Services	Y
Research Property Management	Y
Research Security Services	N
_____ (PCA)	Y
BTRC	Y
Project File	Y
Other _____	N
_____	N

NOTE: Final Patent Questionnaire sent to PDPI.

Summary

During the past six months, we have consolidated many of the concepts contained in the proposal for this grant, and we have begun design and construction of systems to validate those concepts experimentally. An overview of the system design as it currently stands is contained in [McKe84c], attached as an appendix. This is the only paper completed during the period. Several papers are underway, or almost completed. These are described as appropriate below.

Equipment

The National Science Foundation has agreed to purchase test equipment to support our experimentation. This equipment comprises three VAX 11/750's interconnected via Ethernet and a VAX CI. Four well-configured IBM PC's are to be used as interface machines. The total award amount, including Georgia Tech matching contribution, is approximately \$240,000. We are currently negotiating a discount with Digital, the manufacturer. We anticipate delivery towards the end of calendar year 1984. Naturally, our design and construction efforts reflect this timeframe.

Kernel Support Mechanisms

Our work in this area has focused on mechanisms to maintain ordering requirements in the nested action environment. This work is described in a technical report [McKe84a] that is currently undergoing revision before release.

Kernel Design

An operating system kernel to support the operating system is under construction. Initial design concepts are discussed in a technical report [Spaf84] to be released soon. This report focuses on the operation invocation mechanisms, network servers, the virtual memory system, and the file system. Work is now progressing on the construction of several of the low-level components of the operating system.

Compiler Development

In conjunction with this grant, although not funded directly by it, we are constructing a compiler to support system programming for the new operating system. The design of the language to be supported is now substantially complete, and construction of the compiler is underway.

Scheduling Mechanisms

Preliminary work on the design of fault-tolerant scheduling mechanisms began during the past few months. This work is described in an upcoming technical report [McKe84b]. We anticipate that considerable effort will be expended here over the next two years.

User Interface

Work is progressing on the design and construction of a user interface for the operating system. Currently, this involves construction of a windowing operating system for the IBM PC. This will ultimately be used to interface to the VAXen, but immediately we hope to get experience using it in a Unix-style environment.

Publications

Possibly due to the youth of this effort, our publication record during the past six months has not been as good as we would have liked. However, many of the early concepts are now well understood and in draft form. Thus, we anticipate up to five journal submissions and several conference submissions in the next period. The topics to be covered include action management protocols, non-consistent database algorithms, action ordering mechanisms, specification techniques for fault-tolerant jobs, and scheduling mechanisms for fault-tolerant job execution. The last two papers mentioned also address dynamic reconfiguration. Possibly a paper on the file system will be completed during this period also.

The paper attached as an appendix was invited by the IEEE Interest group on distributed computing for a special issue of the newsletter. The issue is to focus on distributed operating systems.

References

- [McKe84a] McKendry, M.S., "Ordering Actions for Visibility," School of Information and Computer Science, Georgia Institute of Technology, Technical Report, 1984.
- [McKe84b] McKendry, M.S. "Fault-Tolerant Scheduling Mechanisms," School of Information and Computer Science, Georgia Institute of Technology, Technical Report.
- [McKe84c] Attached as an appendix.
- [Spaf84] Spafford, E.H., and M.S. McKendry, "Kernel Structures for Clouds," School of Information and Computer Science, Georgia Institute of Technology, Technical Report.

Clouds: A Fault-Tolerant Distributed Operating System*

Martin S. McKendry †

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Introduction

Over the past few years, several research groups have been studying techniques to exploit the potential of distributed systems for supporting fault-tolerance. At Georgia Tech, the *Clouds* group is working on the design and construction of a fault-tolerant, reconfigurable distributed operating system. We view a single operating system as controlling a set of computers called a *multicomputer* or *computer cluster*. Without impacting services provided to users, we intend to provide transparent support for *upward* reconfiguration -- addition of machines to the multicomputer -- and *downward* reconfiguration -- failure or removal of machines. Thus, a fundamental difference of our work from much other work in the area is our attitude toward *autonomy* [McKe83]. We regard autonomy as an issue of operating system policy, not structure. Work is assigned to the machine(s) most able to handle it. In this way, service to all users can be enhanced.

The foundation of our approach depends on a combination of *actions*, *objects*, and fault-tolerant job scheduling mechanisms. Objects are instances of abstract data types. They encapsulate a data part, which represents their state, and a procedural (or operation) part, which specifies the changes that can be made to the data. Actions are partial orders of operations on objects. In considering reliability, the fundamental characteristic of interest is that actions appear to happen completely or not at all (i.e., they support failure atomicity). Since actions cannot actually be implemented atomically, recovery and synchronization mechanisms are provided to give the same effect. Thus, the effects of an action on an object are not made permanent until the action completes (*commits*). A failure prior to this time results in the object reverting to a state consistent with the failed action never having existed. Actions fail (*abort*) when machines with which they have interacted fail and when they are deliberately aborted by users.

Clouds supports objects and (nested) actions as fundamental storage and computation mechanisms respectively. Objects are implemented as components of virtual address spaces. Actions are supported by low-level network protocols that handle lost and duplicated messages, remote procedure calls, and orphan detection. There is a tightly-controlled relationship between volatile (main) memory and permanent memory (usually disk). This *architectural* support provides a sound basis for consistency. That is, it is possible to establish the restart state of any object after a failure. Even if that object is not available to participate in establishing its state, the state is determined by a predefined protocol.

For a system to survive component failures (i.e., tolerate faults), mechanisms are needed in addition to basic action support. First, some representation of *work* is needed. In Clouds, we represent work with *action networks*. Action networks are programmed using a Petri-Net notation [Pete77]. The transitions of Petri-nets correspond to action executions. One of the attractions of this approach is that the state of such a net (called a *job*) is easily defined: actions execute (transitions fire) atomically, and thus the net changes state atomically.

The second requirement for survivability is that continuity of job execution be maintained through failures. To meet this requirement, we use *job schedulers*. There may be multiple job schedulers in a system. A job scheduler assigns actions to machines, and it provides backup in case of coordinator failure during action commit. The location of a scheduler's components bears no necessary relationship to the sites that are assigned to execute the actions within a job. Each *primary* scheduler is supported by some set of *backup schedulers* that monitor the primary scheduler's activity. One or more backup schedulers are needed, depending on the failure modes that must be tolerated by the scheduling mechanism. There may also be a set of *alternate* scheduling machines. These machines will take on scheduling responsibilities if a failure occurs. They may be predefined in order to eliminate the damage caused by certain failure modes, particularly partitions.

* This research is funded in part by NASA grant NAG-1-430 and by NSF grant DCR-8316590

† Electronic mail may be directed to the author addressed as:
Martin @ GaTech (CSNet)
Martin.GaTech @ CSNet-Relay (ARPA)
...!{akgua, allegra, ulyssees, ut-sally}!gatech!martin (uucp)

The mechanisms provided by the architecture and the job scheduler support the operating system itself and the services it provides. Thus, the operating system is fault-tolerant and decentralized. For example, a print

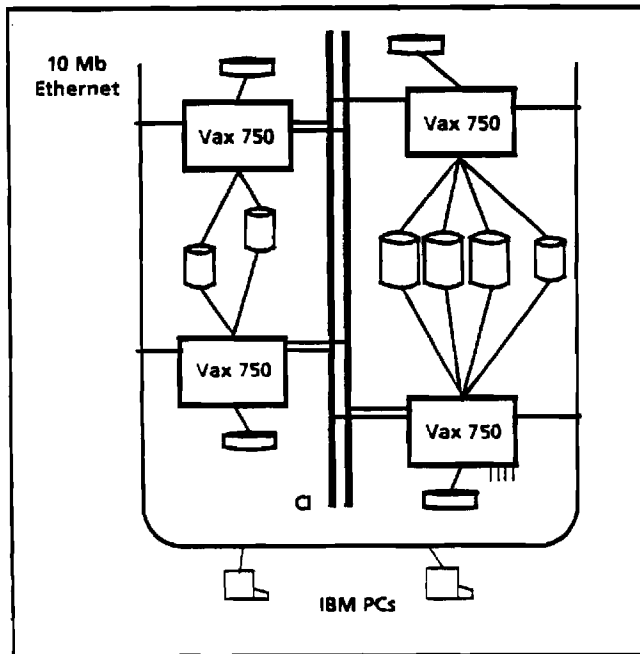
server could reside across multiple machines, and move if one of the supporting machines failed. Similarly, a job scheduler may be regarded as providing a service, and it too will move if one of the machines supporting it fails. The operating system may thus be regarded as supporting a set of resilient distributed objects, or *servers*, that combine to implement a reliable environment to support application systems.

This report outlines the approaches taken in the operating system design to various problems that arise. The report is essentially a collection of abstracts addressing various issues. For more detail, specific reprints can be requested by contacting the author.

Hardware

Our hardware environment consists of computers (machines) connected by relatively high-speed local area networks. In effect, we assume that all computers are located in a single building or vehicle (although many techniques have broader applicability). The complete set of machines, with their connecting networks, forms a *multicomputer* or *computer cluster*. From this multicomputer, we view most interactions with "reality" (i.e., users) as taking place via devices that are accessible from multiple computers within the multicomputer. Devices that do not satisfy this depend on their supporting *host*, and thus their survivability is a function of the host's surviving ability.

Our experimental configuration is diagrammed below. In this configuration, we use VAX 11/750's as the multicomputer machines. These are connected by two networks, thus providing redundancy in the communication system. User access is via personal computers, which are accessed through the Ethernet.

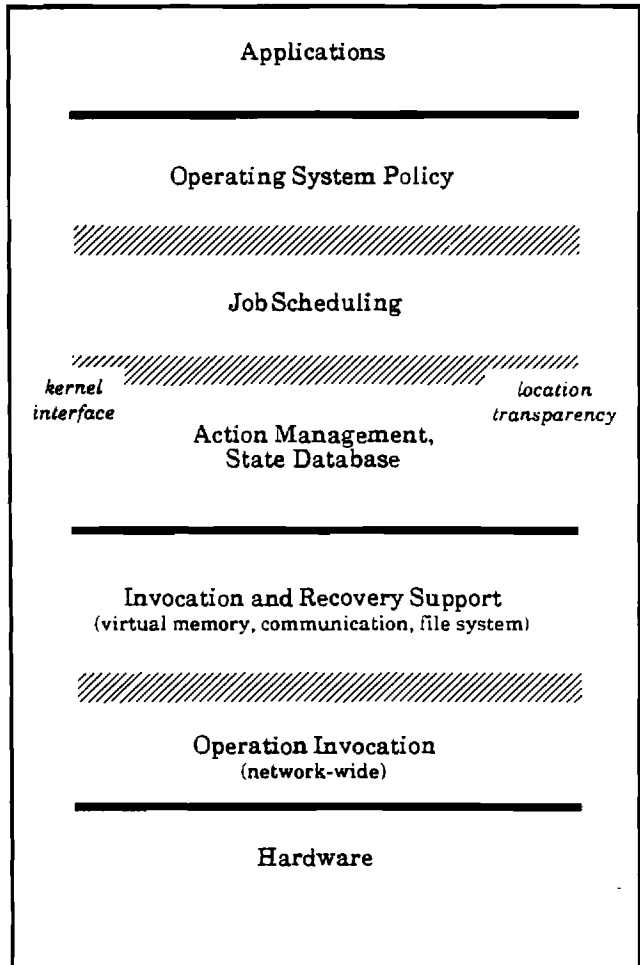


This use of personal computers reflects a secondary investigation into the effects of moving highly interactive tasks, such as editing, to independent processors. By providing this transparently to users, we hope to reduce the load on the multicomputer machines and improve response to users.

Operating System Structure

It is difficult to show an explicit hierarchy of dependencies among operating system components, and thus we cannot claim a strictly level-structured system. (For example, the job scheduler both supports and uses the action management system.) Nonetheless, the approximate hierarchy shown below illustrates the structure of the system.

Important aspects in this structure include the primitive level at which operation invocation is supported and the level at which location transparency is implemented. If the object referenced is local and active, then operation invocation is regarded as if it was a single instruction. Otherwise, a fault occurs and a search is initiated. This provides a high degree of transparency, even to such low-level mechanisms as virtual memory. Paging across the network, for example, requires no special support. Full location



independence is available to operating system policy mechanisms. It is also available to the job schedulers, except inasmuch as the components of schedulers must be placed carefully if the schedulers are to survive failures, and schedulers must be aware of machine locations in order to balance loads.

A distributed *state database* supports many operating system mechanisms. In particular, it supports orphan detection for action management and it supports load distribution by job schedulers. This database makes no guarantees about the consistency of the views it presents. Rather, it guarantees only that all sites will eventually converge to the same view if no additional changes are made. It stores such information as the load on particular machines and data to assist in formulating search strategies.

Object Structure

Objects are passive. An object is represented by an address space component (the P0 address space on the VAX) that is mapped into the virtual memory space of the process (or action) making an invocation. The address space of an object contains three main components. The first contains *permanent* data. This data survives failures, and is maintained in accordance with the action mechanisms. Note that in many operating system objects there is no permanent data. The second component of the object is its *volatile* data. This data is lost on machine failure and is removed when objects are *deactivated*. Objects are deactivated (swapped out to disk) when there are no outstanding actions and they have not been accessed for "a long time". Finally, each object has its code, or *type*, component. This component may be shared with other objects. It represents the structure of the data and the operations which can be executed on that data.

In addition to the standard operations, objects may contain special operations that are invoked by the operating system when action events occur. These events include *BOA* (Beginning of Action), *precommit*, *commit*, and *abort*. Programming these events can be complex. However, there are simple defaults available, and we anticipate that most specialized object programming will be to support the operating system, and thus will be done by systems programmers.

Operation Invocation

Object operation invocations are executed at the machine supporting the data portion of the object. (This is a simplification that would not be maintained if diskless machines existed in the multicomputer.) There are too many objects for all objects to be represented in main memory. Thus, a kernel structure, called the Known Object Table, contains information about recently referenced objects. When an invocation occurs, this table is examined. If the object referenced is known and is local, the invocation proceeds immediately: the object's address space is mapped into the current process's address space, and execution begins at the appropriate operation.

If the object is not known, a search is initiated. If the object is found, the invocation is eventually performed by a *slave* process at the machine supporting the object. In a system such as Clouds, a referenced object could be on any machine and, unless policy considerations implement restrictions, is equally likely to be on any particular system (assuming random distribution). Thus, the search mechanism is critical to the efficiency and success of the approach. It is not possible, for example, to broadcast the search to all machines: in the worst case, they would all have to access local disks several times. We are expending considerable effort to develop fast search mechanisms that terminate as quickly as possible if it is known that an object is not on a particular machine or disk.

Objects can also move. It is therefore a principle of the system's design that the only information that may be treated as correct by any machine's kernel is information about objects on that machine. Thus, every invocation sent to another machine contains an implicit search; the object could have moved since it was last referenced.

Action Support

Processes interact with most objects in the context of actions. Furthermore, kernel calls such as *create* and *delete object*, *move file*, or *change type* are implemented as nested actions, thereby insulating the user from kernel failures. The kernel implements those mechanisms necessary for recovery and synchronization. Synchronization is supported by locks and an ordering mechanism. Recovery is supported by the virtual memory mechanism and an action-oriented file system. We will address these in turn.

To provide conventional action synchronization (i.e., to implement concurrency atomicity), it is necessary to restrict the *visibility* of changes made by actions. In Clouds, we use pessimistic synchronization. For many applications, this can be provided by a programmable locking mechanism [Allc83b]. This mechanism allows the object programmer to name locks associatively. Thus, a lock instance is dissociated from the data structure being locked. This freedom addresses the phantom problem [Eswa76], in the case that no data structure exists at the time locking is required. It also allows locking to be based on logical, rather than physical, constraints. For example, all variables within a symbol table that correspond to a single variable name could be locked with a single lock instance.

Locks control visibility In operating systems it is also necessary to control the order of access by actions to objects and items managed by objects. This is necessary, for example, in producer-consumer and limited resource situations. Conventionally, this is addressed via semaphores or equivalent mechanisms. For Clouds, we have developed a set of mechanisms that implement synchronization for ordering within the action environment [McKe84b]. These mechanisms implement ordering constraints for recoverable items, such as queue entries, and non-recoverable items, such as disk pages. These items

have distinctly different visibility requirements. Furthermore, it is clear that the cost of fully general ordering mechanisms may be prohibitive in the action environment. More restricted visibility, and thus more efficient ordering mechanisms, may be sufficient. As a side benefit, the more efficient mechanisms permit simplified recovery to be used in many situations.

Recovery is supported by a specialized virtual memory system and an action-oriented file system. The virtual memory system recognizes actions by permitting shadow paging between action precommit and commit. Ultimately we hope to use a more efficient system based on intention lists [Gray78]. This approach would provide more efficient support for recoverable items of small granularity, such as queue entries. Our current approach separates the object's view of recovery and synchronization for these items from the virtual memory view. Thus, either the shadow paging or the intention list scheme could be used without changing the object.

Action Management

To implement actions on objects, a protocol is needed to coordinate operations and action events. In Clouds, the action management protocol also incorporates a complete remote procedure call (RPC) mechanism [Nels81, Spec81]. The protocol makes only minimal assumptions about the underlying communication system. We have found that less stringent communication assumptions (e.g., messages, if delivered, will be delivered correctly) require fewer total messages than if reliable transmission is assumed. Thus, the action management protocol handles lost, retransmitted, and reordered messages.

In particular, the protocol satisfies three basic goals:

1. It avoids creating nested actions on every remote procedure call. The result is that remote procedure calls (including ones within actions) are very efficient -- only a single call message and reply message are needed in many cases.
2. It is uniform at all levels -- whether, for example, commitment is of a nested or top-level action.
3. It solves the orphan problem. All actions, including ones that have begun abortion but have not yet completed, see consistent system states.

In essence, the action management protocol implements RPC's within an action environment. Commitment uses a relatively standard two-phase algorithm with extensions to handle concurrent nested actions. It is the first protocol to decouple RPC from creation of nested actions, thus yielding fewer messages on the net.

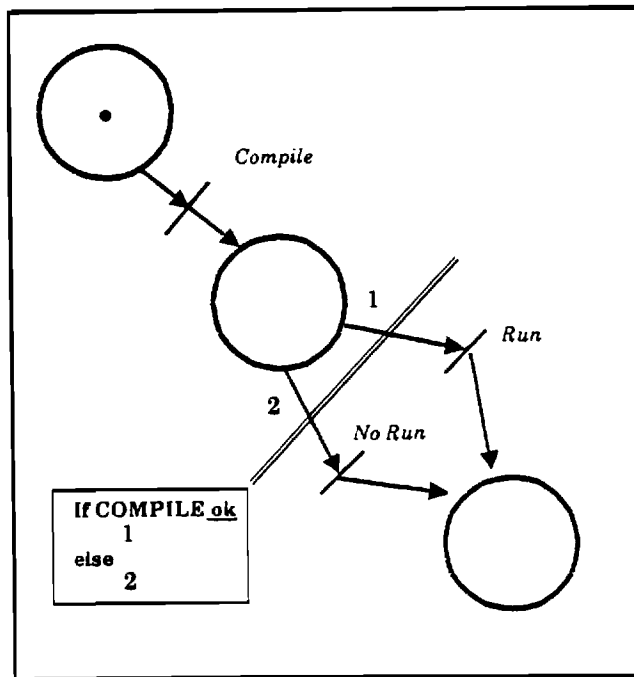
Job Representation

Action support mechanisms and the action management protocol lead to consistency: when a

failure occurs, the state of objects involved in the failed action is defined. If the failure is of a nested action, the parent can take some alternate approach. However, if the failure is of a top-level action, the work represented by that action stops unless additional support is provided. To solve this problem, some representation of work is needed. It must be possible to specify other approaches to getting work done if a failure occurs. To exploit the potential of the multicomputer environment, it should also be possible to submit concurrent top-level actions.

Our approach to work (*job*) representation is based on adaptations to Petri Nets [Pete77]. In this approach, transitions correspond to actions. Actions take a set of tokens as input and produce tokens as output. These tokens may have attributes. For example, a compiler might take a token containing the name of the source file to compile. It might produce a token that specifies the name of the object file, plus an indication whether the compile was successful. This indication can then be used as input to the subsequent *run* action. A compile-then-run-if-successful job is illustrated below. This *job* illustrates the use of *action schedulers*. These are used to determine which transition should be taken when alternatives are possible. They may examine state information about the job, the transition, or the attributes of the tokens.

More complex examples, including a print server and recovery block, are presented in [McKe84a].



Job Scheduling Mechanisms

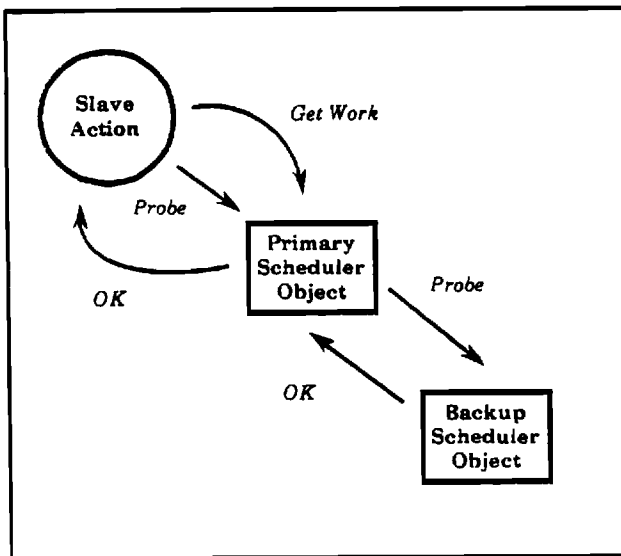
Having specified a job, two problems remain. First, failure of a commit coordinator at certain times will result in a state from which it is difficult to continue without additional support. Secondly, the scheduler itself -- a process, action, or job that schedules enabled

transitions -- might fail. To address these problems, we are developing fault-tolerant mechanisms for job scheduling.

Our initial design addresses single stopping faults only. It also assumes that there is time to reconfigure between faults. The mechanism consists of a set of *slave* actions that require work, a *primary* scheduler object which assigns work to slaves, and a *backup* scheduler object which supports the primary and reconfigures the scheduler after a primary failure. Job representations are stored at both the primary and backup scheduler. The primary selects actions to execute and assigns them to slave actions, depending on which machines support those slaves. It is this selection that controls load distribution. If a slave action fails, the primary scheduler reassigns that work to another machine, possibly after running an action scheduler submitted with the job.

If a component of the scheduler (either primary or backup) fails, the scheduler reconfigures. Intuitively, this is accomplished as follows. The mechanism requires that at all times there be at least one slave that is blocked at the primary scheduler awaiting a work assignment. Each machine using this scheduler has at least one such action waiting. A periodic probe from the machine that initiated the slave action determines whether the primary is still functional. If the probe fails some set number of times, the slave notifies the backup that the primary has failed. The backup then takes over as primary, and creates a replacement backup on another machine. The primary, meanwhile, is probing the backup to ensure that it is still alive. If the backup fails, the primary creates another one.

Each time work (i.e., an action) is assigned to a node, the primary notifies the backup. Furthermore, if the commit coordinator fails, the scheduler will complete the commit. Thus, any component of the scheduler or commit mechanism can fail at any time after precommit processing and the commit will still occur. Thus, the action is completed and the job is able to continue.



Two aspects of this scheduler mechanism are particularly interesting. First, there is no necessary relationship between the machines that support the scheduler and the machines that are assigned work. Also, there is no constraint on the number of schedulers that a system can contain. Thus, a single scheduler could control an entire system, or there might be one scheduler for each "group" of machines. A machine might also take work assignments from more than one scheduler.

The second aspect of note is the manner in which jobs are submitted. Because the primary and backup are implemented as objects, jobs can be submitted within the action system. Thus, a *job submission* job is executed to submit a job. This approach ensures that jobs "arrive" at both schedulers "simultaneously" or not at all.

User Interface

At any time, one machine in the multicomputer is running a job to service each terminal. If that multicomputer machine fails, another machine is assigned the job. Tracking this requires some processing by the user's interface computer. To this end, and to support general user processing, we are developing an operating system for the IBM PC. This system, code-named *Bubbles*, provides a window-management system and supports division of tasks between multicomputer machines and the personal computer.

Programming Languages

Our efforts in programming languages are currently focussed on systems programming. Lacking experience in this environment, we have chosen not to attempt development of an application language but to concentrate instead on providing basic access to system functions. Our initial system is being constructed in C. We are working on a compiler for a system programming language called *Aeolus*. *Aeolus* permits the programmer to capture action events, and even allows portions of an object to be programmed in assembly code. With assistance of the kernel, portions of an object's address space can be flushed to disk on demand. When we have gained experience in the use of *Aeolus* in an operational environment, we will examine application languages. Our hope is that programming paradigms established by system programmers will assist in developing abstractions for application programming.

Related Research

Related research can be categorized in three areas: work on the action/object approach to reliability, work related to objects, and work on operating systems. Work on the action/object approach includes the Argus project at MIT ([Lisk83] and [Wei83]), and the TABS project at CMU ([Schw83]). Argus is focusing on programming languages and techniques, while TABS is an application support system to run in the Spice environment. At the University of Washington, researchers in the Eden project are examining the use

of objects for constructing distributed application systems [Alme84]. There is also work being done there involving the action approach [Jaco82]. At Cornell, the ISIS project is researching fault-tolerant objects based on replication [Birm84].

Some of the notions used in the Clouds operating system kernel are similar to those of Hydra [Wulf74] and other object-oriented operating systems. We share much of our philosophy of operating system design and autonomy issues with the Archons project at CMU [Jens82]. Researchers in Archons are also examining issues related to objects, actions, and increased concurrency [Sha83].

Reports and Publications

The reader wanting more detail can contact the author via either post or electronic mail. Reprints of most of our reports and publications are available, and we maintain a permanent mailing list. Initial motivation is discussed in [McKe83]. An early paper describes approaches to synchronization and recovery [Allc83a]. The action management protocol is described in [Allc83b]. In addition, Allchin's thesis describes locks, recovery mechanisms, and the protocol for the state database (see also [Allc83c]). Mechanisms for controlling ordering are described in [McKe84b], and the scheduling approach is introduced in [McKe84a]. The initial design for the kernel is described in [Spaf84]. Several papers are in progress, and three Ph.D. dissertations on kernel structures, search mechanisms, and programming techniques are anticipated during the next eighteen months.

Acknowledgements

Clouds involves three faculty and many graduate students. Professors Richard LeBlanc and Albert Badre are supervising work in programming languages and user interfaces, respectively. Ph.D. and M.S. students working on the project include Glenn Glover, Kathy Moore, J. Morgan Morris, Dave Pitts, Gene Spafford, Henry Strickland, Bill Thibault, Scott Vorthmann, and C. T. Wilkes. Our progress during the last two years would not have been possible without the enthusiasm of our sponsors and the support and assistance of the administration of the School of Information and Computer Science at Georgia Tech.

References

- [Allc83a] Allchin, J. E., and M. S. McKendry, "Synchronization and Recovery of Actions," Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983
- [Allc83b] Allchin, J. E., "An Architecture for Reliable Decentralized Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1983 (also released as technical report GIT-ICS-83/23)
- [Allc83c] Allchin, J. E., "Using Weak Consistency Conditions," Proceedings of the 3rd ACM-IEEE Symposium on Reliability in Distributed Software and Database Systems, Clearwater, Florida, October 1983
- [Alme84] Almes, G., A. Black, C. Bunje, and D. Wiebe, "Edmas: A Locally Distributed Mail System," Proceedings of the 7th ACM-IEEE Conference on Software Engineering, Orlando, March 1984
- [Birm84] Birman, K., *et al.*, "Implementing Fault-Tolerant Distributed Objects," Cornell University, Computer Science Technical Report 84-594, March 1984
- [Eswa76] Eswaran, K., J. Gray, R. Lorie, and I. Traiger, "The Notions of Concurrency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, No. 11, November 1976
- [Gray78] Gray, J., "Notes on Database Operating Systems," *Lecture Notes in Computer Science*, R. Bayer *et al.*, ed., Springer-Verlag, 1978, pp.393-481
- [Jaco82] Jacobson, D., "Transactions on Objects of Arbitrary Type," Technical Report 82-05-02, University of Washington, May 1982
- [Jens82] Jensen, E. D., "Decentralized Executive Control of Computers," 3rd IEEE International Conference on Distributed Computing Systems, October 1982
- [Lisk83] Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983
- [McKe83] McKendry, M. S., J. E. Allchin, and W. C. Thibault, "Architecture for a Global Operating System," IEEE Infocom, April 1983
- [McKe84a] McKendry, M. S., "Fault-Tolerant Scheduling Mechanisms," Technical Report, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1984
- [McKe84b] McKendry, M. S., "Ordering Actions for Visibility," Technical Report GIT-ICS-84/05, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1984
- [Nels81] Nelson, B. J., "Remote Procedure Call," Technical Report CSL-81-9, Xerox Palo Alto Research Center, 1981
- [Pete77] Peterson, J. L., "Petri Nets," *ACM Computing Surveys*, Vol. 9, No. 3, September 1977, p. 223

- [Schw83] Schwarz, P., and A. Spector, "Synchronizing Shared Abstract Types," Carnegie-Mellon University Technical Report CMU-CS-83-163, Revised November 1983
- [Sha83] Sha, L., *et al.*, "Distributed Co-operating Processes and Transactions," *ACM SIGCOMM* Symposium, 1983
- [Spaf84] Spafford, E. H., and M. S. McKendry, "Kernel Structures for Clouds," Technical Report GIT-ICS-84/09, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1984
- [Spec81] Spector, A., "Multiprocessing Architectures for Local Computer Networks," Ph.D. Thesis, Stanford University, August 1981
- [Weih83] Weihl, W., and B. Liskov, "Specification and Implementation of Resilient, Atomic Data Types," Symposium on Programming Language Issues in Software Systems, June 1983
- [Wulf74] Wulf, W., *et al.*, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, Vol. 17, No. 6, June 1974

A Support Architecture for
Reliable Distributed Computing Systems

Semi-Annual Status Report:
May 9, 1984 - July 14 1985

From:

Georgia Tech Research Corporation
Atlanta, Georgia 30332

To:

National Aeronautics and Space Administration
Langley Research Center

Grant No.: NAG-1-430

Endorsements:

Partha Dasgupta
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572

Richard J. LeBlanc Jr.
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2592

A Support Architecture for Reliable Distributed Systems

1. Introduction

The Clouds Project at Georgia Tech is conducting research aimed at building a reliable distributed operating system. The primary objectives of the Clouds operating system are:

- 1] The operating systems will be distributed over several sites. The sites will have a fair degree of autonomy. Yet the distributed system should work as an integrated system. Thus the system should support location independency for data, users and processes.
- 2] Reliability is a key requirement. Large distributed systems use significant number of hardware components and communication interfaces, all of which are prone to failures. The system should be able to function normally even with several failed components.
- 3] The processing environment should guard against both hardware and software failures. The permanent data stored in the system should be consistent.
- 4] Distributed systems often have dynamic configurations. That is, newer hardware gets added, or faulty hardware is removed. The system function should not be hampered by such maintenance chores. Thus the system should be dynamically reconfigurable.
- 5] The system should be capable of monitoring itself. This encompasses hardware monitoring for keeping track of hardware failures as well as monitoring key software resources (for example daemons, network servers, and so on.) On detection of failure the system should be able to self-heal (restart daemons) or self-reconfigure (eliminate faulty sites).
- 6] The users should be shielded from both the configuration of the system (site independence) as well as its failure modes. For example, if the site a user is connected to fails, he should be transferred to an active site transparently.
- 7] Many of the above functions can be implemented on conventional systems, but would make the system extremely slow. Thus efficiency is an important design criteria.

The above requirements can be handled by a distributed system and have been designed into the Clouds operating system. Most of the functions have been designed into the kernel of the system, without making the kernel too complex, bulky or inefficient. The design philosophies adopted for the Clouds operating system are:

- 1] An object-based, passive system, paradigm is used as the basic architecture. All system functions, data, user programs and resources are encapsulated as passive objects. The objects can be invoked at appropriate entry points by processes.
- 2] The objects in Clouds represent nearly everything the system has to offer. The site independence philosophy is implemented by making the object name space (system names) flat and site independent. When a process on any machine invokes an object located anywhere, no site names are used. Hence the location of any particular object is unknown to a process.
- 3] Reliability is achieved through two techniques. One of them is the action and recovery concept. The action mechanisms are supported at the kernel level. Actions are atomic units of work. Any unfinished or failed action is recovered and has no effect until it completes. The recovery mechanisms are supported inside every object an action touches.
- 4] Reliability is further extended by the self monitoring and self reconfiguration subsystems. This is a set of monitoring processes that use "probes" to keep track of all key system resources, both hardware and software. On detection of failed or flaky components, the monitoring system invokes the reconfiguration system which rectifies or eliminates (if possible) the faulty components, and initiates recovery of affected actions. The monitoring and reconfiguration subsystems are also monitored by the monitoring system.
- 5] The consistency requirements of the data are handled by the recovery mechanisms and by concurrency control techniques. The concurrency control is handled by synchronization paradigms that are an integral part of the object handling primitives. The synchronization of processes executing in an object is handled automatically by semaphores that are a part of the object. This gives rise to a two-phase locking algorithm that is supported by the kernel as a default. The object programmer has the choice of overriding these controls and use custom built concurrency control, depending upon the application. It is also possible to turn off the default recovery and commit strategies.
- 6] Efficiency has been of concern. The object invocation, recovery and synchronization are handled by the kernel. It turns out that these can be done at the kernel level without much overhead. Since the entire Clouds design is primarily based on object manipulations, invocation and synchronization will be the most used operations. Implementing them at the kernel level will result in an efficient system.
- 7] The site independence at the user level is handled in part by using intelligent terminals. The user terminals are not hard-wired into any machine or site, but are on an ethernet, accessible by any site. Each user session is, of course, handled by one

particular site, but any failure causing the controlling site to be unaccessible causes the user to be transferred to another site. This is handled cooperatively by the user terminal and the other sites. Thus the user terminals are actually intelligent microprocessor systems on the Clouds ethernet. In addition to cooperation with the Clouds network, the user terminals run "Bubbles", a multiwindowing, user-friendly interface to Clouds.

2. Progress Report

The following is a brief report of the current status of the implementation of Clouds.

2.1. Equipment

The test equipment for implementing Clouds was funded by the National Science Foundation and has arrived. Three VAX/750 computers interconnected by a ethernet was installed in November 1984. They have been heavily used to develop the Clouds kernel and allied software described later. Three IBM-PC/XT computers, arrived in July 1985 and are being used to develop the intelligent terminal interface to Clouds. One IBM-PC/AT is scheduled to arrive soon and will be used as the primary development system for Bubbles and the ethernet handling code for the terminal interface.

2.2. Clouds Kernel Design

The kernel design has been through several design phases and is nearly complete. The design effort has produced a cohesive set of implementation guides to the entire Clouds kernel.

The current designs are based on assumptions about efficiency and ease of implementation that seem to be intuitively clear at this time. We may have to reiterate some design decisions and modify some strategies after more hard data is available from the implementation experience.

2.3. Kernel Implementation

The kernel has been implemented to a large degree. The process dispatcher, the virtual memory, object invocation procedures, and some storage and communication software has been implemented and tested. We currently do not have software to build Clouds objects, and thus have not been able to test the invocation in a multiprocess environment. The kernel has been tested in a stand alone system with hand-coded objects.

The most important communications package in Clouds, the ethernet driver has been implemented and tested. The driver is based on a very general design and has the ability to support a host of protocols that can be hooked to it. It currently talks to the Clouds machines as well as the machines running Unix 4.2bsd (tm).

The storage management subsystem is partially implemented. Disk drivers for implementing the file system (for object storage) is in the test phase. The advanced virtual memory features needed by Clouds (partitions, object mapping, segment handling) is being coded and tested. Implementation of virtual disks using the ethernet (for inter-site paging) is underway as the ethernet driver is now available.

The next phase will integrate the results of the compiler building (below) with the kernel to allow building of services and user programs as objects and running them on using multiple processes, and multiple sites.

2.4. Compiler Development

The systems programming language for Clouds implementation is Aeolus. Currently all development is being done on a VAX running Unix 4.2bsd, using "C". This is pending the full implementation and testing of the Aeolus Compiler. Once the compiler is implemented, and it interfaces to the Clouds system (the compiler generates objects), further development will use Aeolus.

The compiler implementation for Aeolus is currently underway. The Amsterdam Compiler Kit (ACK) is being used to generate code for both the Clouds system running on VAXen, and the Bubbles system running on 8088/8086 based systems (IBM-PC/XT/AT). Work on semantic routines for Aeolus is proceeding in parallel with the development of intermediate code for ACK. This work is being done in Pastel, an extended Pascal dialect.

2.5. Action Management

The action management is an advanced kernel subsystem that ensures the atomicity of the distributed actions of the Clouds system. The action management is responsible for creation, deletion, proper or improper termination of actions, commitment, and failure containment. The design of the action management subsystem is as far complete as can be achieved theoretically without availability of implementation experience. The implementation of action management will be underway in late 1985, upon completion of the base kernel, and the availability of multiple sites running multiple processes.

2.6. Fault Tolerance and Probes

Use of probes in monitoring and fault tolerance is being studied. Probes are somewhat like messages, but unlike messages they are handled by traps handlers in processes and special probe handlers in objects. Thus probes can be sent to both passive as well as active entities. This gives rise to a powerful paradigm that is useful for a lot of activities, from monitoring, status enquiries to emergency messages. An application of probes to fault tolerant scheduling has been discussed in [McKe84c].

2.7. User Interfaces

The Clouds user interfaces are at several levels. The Clouds system runs a shell that allows hierarchical name spaces and common shell functions as the service routine for each user. The interface to this shell is via the intelligent terminals. This part of Clouds is still under the design phases.

The Human Factors group at Georgia Tech is looking at advanced user interfaces which will use the properties of "transitionality" to handle novice and advanced users at their own levels of sophistication. The transitional user interfaces will be built both at the intelligent terminal level as well as the Clouds shell level.

2.8. Publications

The publications that have resulted from this research has been referenced below.

3. References

[Allc82]

Allchin, J. E., and M. S. McKendry, Object-Based Synchronization and Recovery, Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982

[Allc83a]

Allchin, J. E., An Architecture for Reliable Decentralized Systems, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also released as technical report GIT-ICS-83/23)

[Allc83b]

Allchin, J. E., and M. S. McKendry, Synchronization and Recovery of Actions, Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983

[AlDaLeMcSp85]

Allchin J. E., Dasgupta P., LeBlanc R. J., McKendry M. S., Spafford E., The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System. (draft)

[LeBl85]

LeBlanc, R. J., and C. T. Wilkes, Systems Programming with Objects and Actions, Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, Colorado, May 1985 (also available as Technical Report GIT-ICS-85/03)

[McKe82]

McKendry M. S. and Allchin J. E. Object-Based Synchronization

and Recovery Technical Report GIT-ICS-82/15, Georgia Inst. of Tech.

[McKe83]

McKendry, M. S., J. E. Allchin, and W. C. Thibault, Architecture for a Global Operating System, IEEE Infocom, April 1983

[McKe84a]

McKendry M. S. Clouds: A Fault-Tolerant Distributed Operating System. Technical Report, Georgia Inst. of Tech.

[McKe84b]

McKendry, M. S., Ordering Actions for Visibility, Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)

[McKe84c]

McKendry M.S., Fault Tolerant Scheduling Mechanisms, School of ICS, Technical Report, Georgia Tech.

[Spaf84]

Spafford E. Kernel Structures for a Reliable Multicomputer Operating System Thesis Proposal, Georgia Institute of Tech.

A Support Architecture for
Reliable Distributed Computing Systems

Semi-Annual Status Report:
January 1986 to September 1986.

From:

Georgia Tech Research Corporation
Atlanta, Georgia 30332

To:

National Aeronautics and Space Administration
Langley Research Center

Grant No.: NAG-1-430

Endorsements:

0 1

Partha Dasgupta
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572

Richard J. LeBlanc Jr.
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2592

A Support Architecture for Reliable Distributed Systems

1. Introduction

This report is a brief overview of the activities of the *Clouds* research group, in connection with the funding received from NASA under the grant NAG-1-430. First we state the research goals of the *Clouds* project and then we discuss the current state of the project.

The Clouds Project at Georgia Tech is conducting research aimed at building a reliable distributed operating system. The primary objectives of the Clouds operating system are:

- 1] The operating systems will be distributed over several sites. The sites will have a fair degree of autonomy. Yet the distributed system should work as an integrated system. Thus the system should support location independency for data, users and processes.
- 2] Reliability is a key requirement. Large distributed systems use significant number of hardware components and communication interfaces, all of which are prone to failures. The system should be able to function normally even with several failed components.
- 3] The processing environment should guard against both hardware and software failures. The permanent data stored in the system should be consistent.
- 4] Distributed systems often have dynamic configurations. That is, newer hardware gets added, or faulty hardware is removed. The system function should not be hampered by such maintenance chores. Thus the system should be dynamically reconfigurable.
- 5] The system should be capable of monitoring itself. This encompasses hardware monitoring for keeping track of hardware failures as well as monitoring key software resources (for example daemons, network servers, and so on.) On detection of failure the system should be able to self-heal (restart daemons) or self-reconfigure (eliminate faulty sites).
- 6] The users should be shielded from both the configuration of the system (site independence) as well as its failure modes. For example, if the site a user is connected to fails, he should be transferred to an active site transparently.
- 7] Many of the above functions can be implemented on conventional systems, but would make the system extremely slow. Thus efficiency is an important design criteria.

Most of these issues have been discussed in previous reports and in *Clouds* literature referenced below. We have been designing and implementing system components to achieve all the abovementioned goals.

2. Progress Report

2.1. Kernel Implementation

One of the primary functions of the *Clouds* kernel is to provide a uniform substrate that allows access to object over a distributed object space. The two major subsystems of the kernel, has been completed by two Ph.D. student who graduated recently.

Eugene Spafford graduated in May 1986. His thesis work included the development of the virtual machine support for the operating system; and object support for Clouds. This involved writing the interrupt handlers, schedulers, device interfaces, the system call interface and the synchronization support. To provide object support, the kernel recognizes the object format, and supports a capability based distributed naming system that allows transparent remote as well as local object invocation.

The Storage Management system was implemented by David Pitts who graduated with a Ph.D. in June 1986. His work provides Clouds with a non-volatile segmented object address space that supports the distributed object storage space. The object storage also support reliable commit protocols that will be tied to the action management system, which will support atomic transactions and will provide the first step towards fault tolerance.

2.2. Clouds Environment

The kernel integration which consisted of integrating Spafford's and Pitts' work into a usable kernel was completed in September 1986 by Bernabeu and Khaldi. As of present the kernel allows creation and deletion of user processes, objects stored on disk, virtual object space, that is swappable and a rudimentary user interface that works over the Ethernet to the front end Unix machine.

The design of the action management system has been completed by Kenley who graduated with a M.S. degree in January 1986. Kenley's work lays a broad set of design rules, specifications and guidelines of implementing the action management system. The implementation work has commenced and will be nearing completion in about one years time.

3. Research Plans

The short term plans include the commissioning of the Aeolus compiler for Clouds (the compiler is ready, but the interface to Clouds is not). The compiler will cross-compile Clouds objects on Unix machines, and Clouds will have the ability to bring in objects from Unix as and when necessary. On a somewhat longer term we are planning to integrate Clouds and Unix at several levels in order to make Clouds provide all the useful Unix services as well as the distribution and reliability that is available in Clouds. More details on the research plans will be made available in the form of a research proposal to NASA.

4. References

[Allc82] Allchin, J. E., and M. S. McKendry, Object-Based Synchronization and Recovery, Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982

[Allc83a]

Allchin, J. E., An Architecture for Reliable Decentralized Systems, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also released as technical report GIT-ICS-83/23)

[Allc83b]

Allchin, J. E., and M. S. McKendry, Synchronization and Recovery of Actions, Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983

[DaLeSp86]

P. Dasgupta, R. LeBlanc and E. Spafford, *The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System*, Technical Report GIT-ICS-85/29, 1985, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.

[Da86]

P. Dasgupta, *A Probe-Based Fault Tolerant Scheme for the Clouds Operating System*, Proceedings of the International Conference on Object Oriented Programming Systems, Languages and Applications, Portland, Sept-Oct 1986. (Also available as Technical Report GIT-ICS-86/05)

[DaMo86]

P. Dasgupta and M. Morsi, *An Object-Based Distributed Database System Supported on the Clouds Operating System*, Technical Report GIT-ICS-86/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986,

[Ke86]

G. G. Kenley, *An Action Management System for a Distributed Operating System*, M.S. Thesis. School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as technical report GIT-ICS-86/01)

[LeBl85] LeBlanc, R. J., and C. T. Wilkes, *Systems Programming with Objects and Actions*, Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, Colorado, May 1985 (also available as Technical Report GIT-ICS-85/03)

[McKe82]

McKendry M. S. and Allchin J. E. *Object-Based Synchronization and Recovery* Technical Report GIT-ICS-82/15, Georgia Inst. of Tech.

[McKe83]

McKendry, M. S., J. E. Allchin, and W. C. Thibault, *Architecture for a Global Operating System*, IEEE Infocom, April 1983

[McKe84a]

McKendry M. S. *Clouds: A Fault-Tolerant Distributed Operating System*. Technical Report, Georgia Inst. of Tech.

[McKe84b]

McKendry, M. S., *Ordering Actions for Visibility*, Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)

[McKe84c]

McKendry M.S., *Fault Tolerant Scheduling Mechanisms*, School of ICS, Technical Report, Georgia Tech.

[PiSp85]

D. V. Pitts and E. H. Spafford, *Notes on a Storage Manager for the Clouds Kernel*, Technical Report GIT-ICS-85/02, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[Pi86]

D. V. Pitts, *Storage Management for a Reliable Decentralized Operating System*, Ph.D. Diss.,

School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986,
(Also released as Technical Report GIT-ICS-86/21)

[Sp86]

E. H. Spafford, *Kernel Structures for a Distributed Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as technical report GIT-ICS-86/16)

[Wi85]

C. T. Wilkes, *Preliminary Aeolus Reference Manual*, Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[WiLe86]

C. T. Wilkes and R. J. LeBlanc, *Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System*, Technical Report GIT-ICS-86/12, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, To be presented at the IEEE Computer Society 1986 International Conference on Computer Languages,

[Wi86]

C. T. Wilkes, *Programming Methodologies for Resilience and Availability*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, in progress, 1986.

G-11-111

**A Support Architecture for
Reliable Distributed Computing Systems**

*Status Report, for the periods:
[1] June 9th 1987 to December 8th 1987
[2] December 9th 1987 to June 8th 1988*

From:

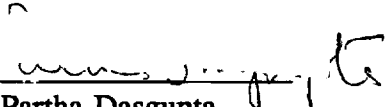
Georgia Tech Research Corporation
Atlanta, Georgia 30332

To:

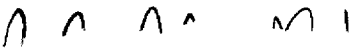
National Aeronautics and Space Administration
Langley Research Center

Grant No.: NAG-1-430

Endorsements:



Partha Dasgupta
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572



Richard J. LeBlanc Jr.
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2592

A Support Architecture for Reliable Distributed Systems

1. Introduction

This report describes the progress under the above funding from NASA. The funding is targeted towards building a integrated distributed operating system using the object model of computation. In this report the research effort will be termed the Clouds Project.

A full description of the Clouds project, its goals, status and directions are described in the proposals and prior reports sent to NASA, as well as several publications, notably a paper in the 8th International Conference on Distributed Computing Systems.

2. Progress Report

The Clouds research team currently consists of five academic faculty, one research scientist and about 8 graduate students. Most of the funding is from the National Science Foundation. The NASA support is used to augment the NSF support and currently it supports one Ph.D. student and some faculty time.

The following sections describe the project results as a whole and not just limited to the NASA funds. We present the progress achieved in the report schedule as well as describe the current status and directions.

2.1. Achievements

This report covers two six month periods. The progress in the first half (i.e. June to December 1987) has been reported in detail in the proposal submitted to NASA in November 1987. The rest of this report details work that has been completed in the period December 1987 to June 1988.

In the past six months, the Clouds project has implemented the Ra kernel and the system object interface. They are described in the following sections.

2.1.1. The Ra Kernel

The Ra Kernel is a completely redesigned minimal kernel for Clouds. Ra runs on Sun-3 computers and is designed to be portable to most virtual memory machines, even multiprocessors.

Ra provides the basic services needed by object based operating systems. In particular, Ra provides low level, short term scheduling, segmented memory management, virtual spaces and light-weight threads. A description of Ra is attached to this report as a supplement.

The Ra kernel has been implemented and is currently operational.

2.1.2. The System Objects

Since Ra is a minimal kernel, it does not provide most of the services necessary in a general purpose operating system. Ra however provides the support needed to plug in the desired services. These services are provided by modules called system objects.

System objects are dynamically pluggable modules that provide the system services. These system services include device handling, networking, object management and so on. Some of the critical system services have been currently implemented and most of the rest is under development.

2.2. Current Status

The research under progress in the Clouds project as of now is the continued development of system objects for support for a variety of services. Some of the notable ones are described below.

2.2.1. The Device Handlers

System objects providing the device services include tty drivers, Ethernet drivers and disk drivers. All these drivers will use a common interface to the kernel called the device class. The implementations of these interfaces and the terminal driver and Ethernet driver is underway. Some prototyping of the Ethernet protocol driver has been completed and the implementation under Ra is in progress.

2.2.2. The User Object System

Ra provides all the tools to build objects, but does not support objects directly. This is done to separate the mechanisms and the policy, as support for objects involve many policy decisions. The object manager is a system object that supports user level objects.

The object manager provides the functionality of creating and deleting objects, provide naming services and manages the consistency needs for the objects. Most of the design work for the object manager has been completed and the implementation is in progress.

3. Conclusions

The Clouds project is well underway to its goals of building a unified distributed operating system supporting the object model. The operating system design uses the object concept of structuring software at all levels of the system. The basic operating system has been developed and work is under progress to build a usable system.

The Architecture of *Ra*: A Kernel for *Clouds**

*José M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi,
Mustaque Ahamad, William F. Appelbe, Partha Dasgupta,
Richard J. LeBlanc and Umakishore Ramachandran*

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Technical Report GIT-ICS-88/25

Abstract

Ra is a native, minimal kernel for the *Clouds* distributed operating system. *Clouds* supports persistent virtual spaces, called *objects* and *threads* that direct computations through these spaces. *Clouds* is targeted to run in a compute-server — data-server environment, in which processing is done by threads executing on compute-servers (or workstations) and the objects will permanently residing on data-servers (also called file-servers). Service between data servers and workstations is handled by a *distributed shared memory* paradigm.

The major function of *Ra* is to provide segment-based virtual memory management, and short term scheduling. *Ra* provides support for objects through a primitive abstraction called *virtual spaces*. Virtual spaces are composed of *windows* on *segments*. *Ra* provides support for threads through *isibas*. An *isiba* is the unit of activity supported by *Ra*.

The *Clouds* system will be supported on top of *Ra*, by adding *system-objects* to *Ra*. System-objects provide services, such as communication, thread management, atomicity support, object naming and location, and device handling. *Ra* provides mechanisms that allow system-object to be plugged in and out of *Ra* dynamically. *Ra* also provides a set of kernel classes that allow machine dependent variables and interrupt services to be accessed from system objects in a uniform machine independent fashion. Thus *Ra* is a highly modular, expandable and flexible kernel.

Ra is currently being implemented on a collection of Sun-3 workstations and will eventually run on a heterogeneous network of computers of varying sizes including multiprocessors.

This paper describes the architecture and functionality of *Ra*, preliminary details on its implementation, and the envisioned strategy for building a distributed operating system (*Clouds*) on top of *Ra* (by adding appropriate system-objects).

*This work has been funded by NSF grant CCR-8619886.

1 Introduction

This paper is a description of the architecture of *Ra*. The motivation for this work comes from the need for a robust kernel that can support the development of a distributed object-based operating system called *Clouds*. *Clouds* is a research project at the Georgia Institute of Technology. The goal of the *Clouds* project includes the development of a uniform, integrated computing environment (OS and applications) based upon the object paradigm (see below). *Clouds* is the operating system for this environment.

The first kernel for *Clouds* was a monolithic kernel. This prototype kernel executed on a set of minicomputers (Vax-11/750) and has been operational since 1987. As we enter a new phase of the project we have felt the need for a more robust, simple, portable and extensible kernel. *Ra* is the result of a fresh design attempt, using the minimal kernel approach. The design of *Ra* reflects the experience gained and lessons learned during the design, implementation and use of the original *Clouds* kernel [Spa86,Pit86].

1.1 The Minimal Kernel Approach

A *minimal* kernel converts the underlying hardware into a small set of functions (or abstractions) that can be effectively used to build operating systems. Any service that can be provided outside the kernel (without adversely affecting the performance), is not included in the kernel. Minimal kernels normally provide memory management, low-level scheduling and mechanisms for inter-entity interactions such as message-passing or object-inocations. The minimal kernel idea has been effectively used to build message-based operating systems such as the V-system [CZ83], Accent [RR81], Ameoba [TMS1].

The minimal kernel method of building operating systems is akin to the RISC approach used by computer architects. Some of the advantages of minimal kernels are:

- The kernel is small, hence easy to build, debug and maintain.
- The separation of mechanisms from policy. The kernel provides the mechanisms, and the services above the kernel implement policy [WCC*74].
- The functions provided by the kernel are expected to be efficient (fast).
- Most of the services of the operating system are implemented *above* the kernel, hence they are simpler to test, debug and maintain. The services can also be added, removed or placed without the need for recompiling the kernel, or in most cases without the need for rebooting the system.

- The same operating system can support a variety of services or policies (or even several instances of the same service) which is not generally possible in monolithic kernel situations.

A problem with minimal kernels in message-based operating systems has been performance. If a single server process provides a particular service, there is a lack of concurrency and hence the service waiting times become large. So multi-threaded servers are often used, at the expense of more complex server code. This problem does not exist in monolithic kernels (such as Unix), since the kernel services are protected procedure calls, and can be executed concurrently by multiple processes.

In an object system like Ra/Clouds, the services above the kernel reside in objects and are called upon by user processes like the protected procedure calls to systems services (not via interprocess message exchange). Hence services can be accessed concurrently by multiple clients. However object-invocation overhead is higher than the system-call overhead, therefore there is some degradation in performance compared to a monolithic kernel, but this is less than the cost incurred in a message kernel, as well the benefit of simpler code in the service routines. Also, the design of Ra maps the system object (system services) address space into the kernel address space to substantially reduce the service object invocation cost. Thus Ra achieves most of the benefits of a minimal kernel and does not suffer from some of the performance (or complexity of server) penalties.

The rest of the paper is organized as follows. First we present a brief overview of Clouds. Then in Section 3 we present the design goals and objectives of Ra, the rationale for the minimal set of primitives supported by Ra and the hardware needs for implementing Ra.

Section 5 provides details about the primitives supported by Ra, and section 6 shows the mechanisms used for providing extensibility in the Ra design. Finally section 8 shows how the Ra primitive abstractions will be used to build the Clouds operating system.

2 The Clouds Operating System

The design of Clouds has been influenced by the following concepts, that are basic to it.

- Clouds is a distributed operating system.

The hardware environment for Clouds consists of a set of one or more computers connected by a network. The set of computers can be a collection of mainframes, or more desirably, a

set of compute-servers (or workstations) and a set of data-servers (or file-servers). Clouds integrates all the sites on the network to form a single computing resource.

- Clouds uses the *object-thread model*.

The unit of storage and addressing in the Clouds model is an *object*. The unit of activity in the Clouds model is a thread. The Clouds system provides a storage system that is a true *single-level store* [Mye82]. This paradigm is detailed in section 2.1 and in [DLA88].

- Clouds supports consistency of data.

Since all the data is contained in permanent, shared address spaces, failures can destroy the consistency of the data. The integrity of these address spaces are ensured by mechanisms (such as atomic actions) that guarantee consistency of the data [CD87].

2.1 The Object-Thread Model

All data, programs, devices and resources on Clouds are encapsulated in entities called objects. An object consists of a named (virtual) address space, and its contents. A Clouds object exists forever (like a file) unless explicitly deleted.

Threads are the active entities in the system, and are used to execute the code in an object. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. Threads can span objects, as well as machine boundaries. A thread executes in an object by entering it through one of several entry points, and after the execution is complete the thread leaves the object. An object invocation may pass parameters to the callee, and return parameters to the caller. Several threads can simultaneously enter an object and execute concurrently.

The structure created by a system composed of objects and threads has several interesting properties. First, all inter-object interfaces are procedural. Object invocations are equivalent to procedure calls on modules not sharing global data. The modules are permanent. The procedure calls work across machine boundaries. Second, the storage mechanism used in object-based systems is quite different from that used in conventional operating systems. All data and programs reside in permanent virtual memory, creating a single-level store.

The single-level store dispenses with the need for files. Files are special cases of objects, and can be implemented (by the user) if necessary. Similarly the shared nature of the objects provides system-wide shared memory, dispensing with the need for messages. Messages can be implemented by the user if desired.

A programmer's view of the computing environment created by Clouds is the following. It is a simple world of named address spaces (or objects). These objects live in an integrated environment, which may have several machines. Activity is provided by threads moving around amongst the population of objects through invocation; and data flow is implemented by parameter passing. The system thus looks like a set of permanent address spaces which support control flow through them, constituting what we term *object memory*.

To support the Clouds operating system, each compute engine in the network has to run a specialized kernel that supports mechanisms needed to manage objects and threads. The Ra kernel is a minimal kernel designed for Clouds.

3 The Ra Kernel

As mentioned earlier, the design of Ra was motivated by the need for a modular, extensible, maintainable and reliable kernel. To this end we have identified a set of design goals for the Ra kernel.

3.1 Ra Design Goals

The following are basic criteria that have been used in the design of Ra:

- Ra must be a minimal kernel.

This decision was based on experience with testing and maintaining the first Clouds kernel.

- Ra should provide mechanisms to support the Clouds model of computation.

Needless to say, that is the goal for Ra. Ra itself does not necessarily support objects and threads, but provides the mechanisms needed to do so.

- Ra should be extensible.

Ra is intended to be used in a academic/research environment, which implies that it will go through many upgrades, additions of functionality, changes of techniques and so on. The design of Ra should incorporate ease of extension and ease of modification. To this end, our design uses kernel classes, plug-in system objects, and well defined interfaces inside the kernel, that enhance extensibility.

- Ra should be implementable on a variety of computers.

In addition to the design goals, Ra has to meet the following longer term goals to satisfy current plans for Clouds as well as future research interest. Note that the following functions are not supported in Ra, but the design of Ra should allow the following to be supported.

- Ra should provide mechanisms needed to support distributed shared memory.

Ra is targeted to work in a compute-server – data-server configuration. To handle object invocation in such a network, we have designed a technique called *distributed shared memory* [RAK87]. Distributed shared memory provides a means of viewing objects as existing in a single global distributed shared memory, where although the objects reside on a set of data-servers, they can be used anywhere on demand.

- Ra should provide mechanisms needed to support inheritance.

Inheritance has emerged as a useful means of controlling complexity in object-oriented programming languages. We have identified a means of efficiently implementing multiple inheritance by taking advantage of “multiply mapped” virtual spaces.

- Ra should provide mechanisms needed to support resource location.

Resource location has been identified as a critical issue to the performance of a distributed system such as Clouds [Ber87]. Ra must support a clean separation of mechanism and policy to allow implementation of a variety of resource location policies.

- Ra should provide mechanisms needed to support action management.

Controlling consistency and concurrency of access to shared data is needed in the Clouds system. Action management needs support from the kernel, and the design of Ra should provide these functions. These include cooperation with object invocation, detection of read and write accesses to virtual memory, synchronization, and a means for constructing recoverable storage.

- Ra should provide mechanisms needed to support replication.

While considering the incorporation of replication under Clouds, we realized that efficient replication schemes required modifications to the basic object invocation mechanism. Similarly, implementing quorum-consensus schemes [Gif79] was best done by overriding Clouds’ location transparency.

3.2 Ra Primitives

For an effective minimal kernel design, the set of kernel functions should not need constant revision, but should be able to support all the services required by the user level through properly

implemented system service routines.

Given the following set of facilities required of Ra, we can derive the primitive abstractions that Ra should support.

- Ra should support the invocation mechanism since the basic activity in Clouds is object invocation.
- Ra should provide methods for supporting objects as objects are the most important entity in Clouds.
- Ra should provide support for threads, and support for scheduling since activity in Clouds is handled by threads.
- Ra should provide simple yet powerful virtual memory support mechanisms, since objects storage rely heavily on virtual memory.

The need for computation leads to the need for processes and process scheduling. The object invocation support needs the ability to map memory spaces into process spaces. The object support needs the ability to map a collection of memory spaces into a single address space.

In the design of Ra primitives, the basic unit of memory is a *segment*. A segment is used to hold data (or code) as an uninterpreted sequence of bytes. Some segments are permanent (data and code) and some are not (stacks, parameters). The permanent segments reside in *partitions*. Partitions reside on data-servers and the segments are provided by the partitions on demand. Thus, although Ra knows about the existence of partitions, partitions themselves are not part of Ra.

Segments can be coalesced together (using a mechanism called *windows*) into an address space, called a *virtual space*. Each object is supported by one virtual space. The virtual space can live in any one of at least 3 hardware defined segments (or *hardware-segments*, see section 3.4). We call these hardware-segments the *K-space* (for resident objects) the *P-space* (for process specific data) and the *O-space* (for object virtual spaces).

The basic services provided by Ra include the mapping mechanisms needed to provide support for segments, windows, and virtual spaces; and the mechanisms needed to provide support for computations to access the required segments; and the demand paging of segments from partitions.

In addition to the segment and space services, Ra provides support for lightweight processes called *isisbas*. An isiba is scheduled by the low-level scheduler (using time-slices) and can execute

code in any space to which the isiba is assigned. The assignment of isibas to virtual spaces form the basis of object invocation mechanisms.

The attributes and support mechanisms for segments, windows, virtual spaces and isibas are discussed in Sections 5 and 6. Section 8 shows how these abstractions are used to form the final support mechanisms for Clouds objects and threads.

3.3 Ra System-Objects

In addition to the basic services defined above, Ra needs to provide interfaces for other operating system services. Ra does not provide most of the services that a general purpose operating system provides, especially when these services require policy decisions. What Ra does not provide include naming support, communication support, partitions, action management, replication support, and accounting support.

Ra provides a mechanism, by which operating systems services can be implemented via *system-objects*. A system-object is like a Clouds objects in most respects except that it lives in a protected address-space, often is resident (as opposed to paged) and has access to some of the data structures defined by Ra. Unlike the other Clouds objects, the state of a system-object need not survive system crashes, and are re-initialized at reboot. The system-services can be implemented as system-objects.

Like Clouds objects, system-objects can be invoked by user processes, as well as by Ra itself. Thus Ra can use hooks provided by system-objects to get access to a particular service if that is available. For example, management of segments may differ depending on whether the segment is a “recoverable” segment or not. For recoverable segments, the action manager provides hooks that Ra uses to page the segment in and out.

Some system-objects are mandatory, since Clouds cannot function without these. Examples include the partition handlers, the communication drivers and the tty drivers. Most system-objects are optional, such as naming services, action management, accounting, date and time handlers.

Ra is designed to allow plug-in system-objects. That is, system-objects can be loaded or removed at runtime. The system-objects are not linked to the kernel, the binding is dynamic. More details on system-objects are in Section 6.

3.4 Ra Hardware Architecture Assumptions

Ra is targeted to run on a variety of commonly available hardware, especially workstations. For efficient implementations, we assume that the hardware should have virtual memory and networking support. The characteristics of Ra's minimal hardware support are:

- *A large, linear, paged or segmented virtual memory.* In object-based systems computation proceeds through object invocations. An implementation typically requires the unmapping of an object space and the mapping of another for each object call and return. Thus, some form of paged or segmented virtual memory scheme is vital for the efficient implementation of Ra.

- *Three distinct machine-level virtual spaces which may be efficiently manipulated separately.*

The hardware supported virtual address space of the CPU should consist of (at least) three distinct spaces, that we call the K , O and P spaces, for kernel, object and process, respectively (Figure ??). The kernel and system objects are intended to be mapped into the K space, the current process is intended to be mapped into the P space, while the current object is intended to be mapped into the O space. With the exception of the K space, different virtual spaces will be constantly mapped and unmapped into the P and O spaces (Figure ??). In general, the hardware virtual space can conceptually be divided into three segments by appropriate use of the the high-order bits in a virtual address. However this can lead to very large page table. Thus the hardware needs to handle segmented virtual spaces.

- *User/system privileges and page-level read and write protection.*

Most architectures provide at least user and supervisor execution modes and the ability to protect memory, based on the current mode and the operation attempted. The ability to apply a protection specification to a range of memory will also be useful. Page level read write protection, though not mandatory, will allow us to automate the locking support needed by action managers. The detection of a fresh access to a page of data by a thread can be detected and analyzed (read access or write access) and thus set read or write locks as appropriate. If this facility is missing, language processors have to generate code for locking data pages.

4 Related Work

The design of Ra covers two areas of active research, that is, the development of object-based systems and the development of minimal kernel based systems. Some of the notable projects that have implemented object based operating systems are Eden [ABLN85], Cronus [STB86], and Argus [Lis83]. All these systems structure the kernel as a process running on top of Unix. Thus the kernel in these systems are neither native nor minimal.

The Alpha kernel [J*85] used by the ArchOS project and the Clouds kernel [Spa86] are some of the native kernels used in object-based operating systems. Neither are minimal kernels.

Minimal kernels have been used quite extensively to build message-systems. Some of the notable examples are the V-system [CZ83], Ameoba [TM81], Accent [RR81], and Quicksilver [HMSC87].

5 The Ra Primitive Abstractions

The four primitive abstractions recognized by Ra are *segments*, *virtual spaces*, *isibas* and *partitions*. The Ra primitive abstractions reflect the mechanisms needed by an object-based system; that is segments reflect logical data units, virtual spaces reflect larger modules of data and code, isibas reflect computation and partitions reflect long-term storage.

In the object model of computation a single thread of activity spans many addressing domains. Ra dissociates the concept of an address space from the concept of a computation by the distinction between the virtual space and the isiba. Similarly there are many forms of data that are used in the domain of one virtual space. Examples are executable code, stacks and heaps, logical clusters of data (or lockable units), parameters. This leads to structuring of the virtual space as a set of segments. Closely associated with the assembling of segments into virtual spaces is the concept of windows. A window allows us to map a range in a segment to a range in the virtual space, giving a completely general composition mechanism. The non-volatile nature of most segments needs a repository for them. This is provided by the partition abstraction.

5.1 Segment

Segments are the basic building blocks provided by Ra. A segment is an uninterpreted, ordered, set of bytes. Bytes in a segment are addressed by their displacement from the beginning of the segment, and the segment is identified by a system-wide unique *sys-id*.

Segments are explicitly created and persist until explicitly destroyed. They may be of arbitrary length and their size may vary during their lifetime. A segment has a set of attributes, which include length, types, and storage methods. Certain attributes are considered *immutable* and remain as set throughout the life of the segment. Other attributes such as length, and whether a segment is sharable, are considered *mutable* attributes and may be altered during the life of the segment.

One major characteristic of a segment is its *storage* attributes. These attributes will make guarantees about the behavior of accesses to the contents of a segment in the presence of failures. For example, a *volatile* segment is guaranteed to have zero-filled contents on first access, following a failure of the system that hosts the segment. Changes to the segment are guaranteed to persist up to the first system failure following the modification or the next modification. A *recoverable* segment guarantees that changes persist across system failures. Thus segments with attributes refer to particular methods of using hardware to support storage.

In addition to providing naming and addressing for bytes in a single-level store, segments may be shared between virtual spaces. That is, a given location in a segment may be simultaneously mapped by two or more virtual addresses so that concurrently executing isibas may share a common store. Activities may then communicate using any agreed upon protocol through this shared store. Some variations of sharing are possible and these are described in Section 5.2.

A special form of segment called a *fast segment* requires minimal partition support and is highly optimized. Fast segments are volatile and may not be shared. Because of these restrictions, they may be accessed very efficiently and are used in situations where high performance is required. One common use of fast segments is for passing parameters during object invocation.

5.2 Virtual Space

Ra *virtual spaces* are an abstraction of the notion of an addressing domain and provide a metaphor for manipulating virtual memory mapping mechanisms. Conceptually, a virtual space is a collection of sub-ranges of a set of ordered segments.

A Ra virtual space is implemented by a segment called the *virtual space descriptor* which contains a collection of *windows* (Figure ??). Each window identifies a segment, a range of virtual addresses in the space, and locations in the segment that back the designated virtual addresses. The term window emphasizes that ranges of virtual addresses need not map to an entire segment but may map to portions of segments. Windows also describe the protection characteristics of ranges of the virtual space such as *read-only* or *read-write*.

Note that protection is a characteristic of a virtual space and not of its associated backing

segments. Thus, a particular segment may be read-write accessible when it is associated with one virtual space, but read-only when it is associated with another virtual space.

A virtual space is composed (or built) by a sequence of *attach* operations on segments. Attaching a segment to a virtual space associates a range of addresses in the virtual space to a range of offsets in a segment. The attach operation requires as parameters a virtual space descriptor, a segment-id, a length (of the region to be defined by the association), a starting virtual address in the space, and a starting offset into the segment. The attach operation defines a one-to-one mapping between virtual space addresses and segment locations such that virtual addresses starting at the designated address are associated with segment locations starting at the offset and continuing for the length specified. That is, if virtual address x is associated with segment offset y , then virtual address $x + 1$ is associated with segment offset $y + 1$. Note that a virtual address range defined by an attach operation must fall entirely within the specified segment.

The mapped ranges in a virtual space may not overlap, but ranges in the virtual space may remain unmapped. Thus, a single virtual address may not resolve to two or more segment locations but virtual spaces may have “holes” in them. Virtual spaces may share segments (or ranges of locations in segments). No synchronization or access control is implied by an attach operation. Also, a segment may be mapped more than once to a single virtual space; that is, two or more windows in a single virtual space may refer to a single segment. Multiple mapping can happen in two ways: either two or more windows in the space refer to disjoint regions of a single segment, or two or more windows in the space refer to overlapping regions of a single segment. In the second case, distinct virtual addresses in the same space resolve to the same segment location.

The *detach* operation undoes the work of *attach* and removes the association between a range of virtual addresses and range of offsets in a segment.

The *activate* operation on virtual spaces conceptually provides the kernel with the information in the virtual space’s descriptor. Since the window descriptor is stored in a segment, the virtual space activate operation can be described in terms of operations on segments. A virtual space is *activated* by supplying the kernel with the sys-id of the segment that contains the virtual space descriptor. The kernel then communicates with the segment’s controlling partition (see section 5.4) to activate the descriptor segment if necessary and then attach it to the kernel virtual space. In this way, the descriptor’s contents become visible to the kernel. Once the descriptor is attached, some of its information is stored redundantly in kernel data structures for efficiency.

Activate may be seen as a notification to prepare for further activity on the virtual space.

When a virtual space is activated, the kernel allocates segment and page map tables in anticipation of the virtual space being mapped into the hardware P or O spaces. These structures are initialized according to the window descriptions in the virtual space descriptor. Attach and detach operations will cause these mapping tables to be updated as appropriate. A virtual space must be activated before operations such as attach and detach may be performed on it. Such operations will activate the virtual space implicitly when required. The companion operation, *deactivate*, updates the descriptor to reflect any changes necessary, detaches the descriptor segment, and deallocates segment and page maps used by the virtual space.

Install, maps the virtual space into the P or O space. If the specified virtual space has been activated, then the kernel simply needs to update the hardware segment and page tables to reflect the corresponding tables already constructed and maintained by the kernel. If the specified virtual space has not already been activated, the kernel will activate the space, and then install it. Segments may continue to be attached and detached while a virtual space is installed. In this case, the attach or detach operation will update the virtual space descriptor as well as modify the hardware mapping tables. A segment is attached *dynamically* when it is attached to an installed virtual space, and *passively* when it is attached to an uninstalled space.

There are three varieties of virtual spaces, depending upon where a virtual space is installed. The virtual space which maps into the hardware K space is called the *kernel space* or *kernel*. Similarly, a virtual space which is mapped into the P space (or which will be mapped into the P space) is called a *process virtual space*. A virtual space which maps into the O space is called an *object virtual space*.

5.3 Isiba

Ra *isibas* are an abstraction of the concept of computation or activity and are intended to be the lightest-weight unit of computation.

An isiba may be assigned a virtual processor by the operation *install*. The processor is relinquished when the *remove* operation is performed on the isiba. An isiba is said to be *active* after it is installed and before it is removed; otherwise, it is *inactive*. Isibas are composed of two segments: a *context* segment and a *stack* segment. Each segment contains data describing the activity of the isiba it composes.

The context segment contains the kernel accessible state information describing the computation such as the program counter and machine register contents. These values can be accessed as attributes of the isiba. When an isiba is active, some of its attributes may not reflect the current state of the isiba, but only a recent state. These attributes, such as the values of the

machine registers, are called *transient*. When an isiba is inactive, all attributes represent the most recent state of the isiba. (This is analogous to the PCB of a process in a conventional kernel).

An isiba's stack segment contains the call stack of the computation. The stack is represented by a separate segment because it is a user accessible component of the isiba. Placing the isiba's stack in its own segment also allows a simple implementation of the model of computation in which a single activity may execute in multiple address spaces. A distinct stack segment is allocated to the isiba for each virtual space in which it executes. More than one stack segment may exist at any instant if the transfers from one virtual space to another are viewed as nested traversals. However, only one stack segment is considered the *current* stack of the isiba.

Like segments, isibas are uniquely identifiable by a *sysid*. Isibas have four required attributes, *scheduler*, *privilege*, *O space*, and *P space*. The scheduler attribute identifies a scheduler system object to invoke when a high-level scheduling decision concerning the isiba must be made. Privilege is a transient attribute that describes the processor mode (kernel or user) in which the isiba is executing. Privilege is used to define resources accessible to the isiba at any given moment. Finally, each isiba has two associated space attributes which identify the virtual spaces which must be mapped for the isiba to run. Either or both of these attributes may be null. If both attributes are null, then it is assumed that the isiba will execute entirely within the kernel address space (K space) which is always visible and need not be mapped for the isiba to run. Typically, the context segment of the isiba will be mapped into the kernel space and the stack segment into the P space; however, other schemes are possible.

When an isiba is installed, two parameters are provided to the kernel: a time quantum, and a maximum execution time. The isiba is placed on a queue along with other isibas waiting to execute (the *ready* list). A kernel entity known as the *short-term scheduler* (STS) periodically receives clock interrupts. On each interrupt, the STS determines if the currently executing isiba's time quantum has expired. If the isiba's time quantum has not expired, it continues execution. Otherwise if the cumulative execution time is less than the maximum, it is enqueued on the ready list once again and the STS removes the isiba at the head of the ready list and prepares the processor for it to run. If the cumulative execution time of an isiba has exceeded its maximum execution time, the STS calls the high level scheduler associated with the isiba, which will decide what to do with it. This procedure, called *dispatching* the isiba, may require the kernel to remap the P and O spaces. If the newly dispatched isiba has the same P and O space attributes as the previous isiba then no remapping is required. In this case, the two isibas are said to be executing *concurrently* within the same address spaces. If both space attributes of the new isiba are null then it runs, by default, in the kernel address space. The stack and context segments of the isiba must be attached to one of its associated spaces for it to execute.

Note that the space attributes may change frequently while the isiba is executing.

An isiba may be removed by another isiba executing with appropriate privileges or an isiba may remove itself. In either case, the destination of the removed isiba must be specified. Typically, this will be a queue representing a synchronization variable or event. When an isiba is removed by another isiba it may only need to be removed from the ready queue. If the isiba removes itself (i.e. *blocks*) or if the isiba is removed by a concurrently executing isiba on a multiprocessor implementation of Ra, then another isiba must be dispatched in addition to the designated isiba being removed.

Isibas may be used as demons within the kernel or they may be associated with a virtual space to implement “heavier” forms of activity such as Clouds threads. Using isibas to implement threads is illustrated in Section 8.

5.4 Partitions

Segments are primitive in Ra but supporting certain segment attributes such as recoverability may require extensive policies. These policies should be kept out of the kernel. Also, it is not possible to predetermine all segment attributes. Thus we need a mechanism for handling attributes which might change as the operating system evolves. For this reason, a segment handler and repository called the *partition* is introduced. The partition is responsible for realizing, maintaining and manipulating segments. Although, partitions are a Ra primitive abstraction, and the existence of partitions is crucial to the operation of Ra, the partitions are not part of Ra. A partition handler is a Ra system-object. If the partition is stored at a remote site (data-server) the server runs a cooperating part of the partition code, to get the segment.

Each segment is maintained by exactly one partition and a segment is said to *reside* in that partition. Initially, a segment resides in the partition in which it was created. The partition in which a segment resides is referred to as its *controlling* partition. All operations on a segment with the exception of accessing its contents are performed by its partition and are initiated by requests to the partition.

Several operations are possible on segments via their controlling partitions. Segments may be created and destroyed and they may have their mutable attributes changed as described previously in Section 5.1. Segments may also change their partition of residence, that is, they may *migrate*. Migration may be desirable to improve the locality of reference to the segment and its contents. Finally, segments may be explicitly *activated* and *deactivated*. Activating a segment prepares the controlling partition for further activity relating to the segment and returns summary information about the segment and its attributes to the calling entity (typically the

kernel). Deactivating the segment informs the partition that further access to the segment is unlikely in the near term. Reading and writing the contents of a segment are considered operations on the associated virtual space.

Alternative storage hardware such as write-once optical store can be easily integrated into Ra by introducing an appropriate segment attribute and by implementing a partition to support that attribute. Partitions are themselves implemented as system-objects. References to device drivers that access physical storage are encapsulated in partitions.

Segments may also be *remotely activated*. Remote activations are managed by a local partition called the *mediating* partition. The mediating partition receives the activation request and then cooperates with a remote partition to activate a copy of the segment locally. Segments may be shared remotely as well as locally and thus there may be more than one mediating partition for each segment.

Since segments are named by a system-wide unique sysid, remote activations require that the desired segment first be *located*, that is, the node actually containing a permanent copy of the segment be identified. (Typically, each partition is itself associated with either a node where a permanent copy of its segments is to be found or a method for consulting all or a subset of the nodes containing permanent copies of segments). This allows the system to use different location methods depending on the characteristics of the segments to be located (for a more thorough discussion of the performance issues related to resource location in distributed systems see [Ber87,ABA88,AAJK87]). A policy-making object called the *partition manager* may be consulted to determine the appropriate partition to be used for a particular activation request.

Distributed shared memory [RAK87,LH86] is implemented in partitions that support segments with attribute *dsm*. It is an extension of the shared memory model to distributed systems. A set of operations are provided on shared segments and the partition maintains the state of the shared segment through a coherence protocol. The partitions take on the responsibility of acquiring the current copy of a segment on demand.

Distributed shared memory is the mechanism used to run Clouds in the workstation environment. Instead of sending the computation to a remote node, the required segments are brought from the remote node to the local machine. The remote node is the repository of the object (or the set of segments).

6 Mechanisms for Achieving Extensibility

Ra *system-objects* provide structured access to kernel facilities which are grouped into collections or modules called *kernel classes* (Figure ??). The operations available on the Ra kernel classes collectively constitute the Ra *virtual machine*. Ra's system-objects and kernel classes together form the basis of Ra's claims of extensibility. This section describes system-objects and kernel classes, and how they will be implemented, and provides examples of their use.

6.1 System-Objects

Certain system-objects are designated *essential*. Essential system-objects provide services that are necessary for the system to function yet that admit to a variety of implementations. Virtual memory page placement and replacement policies are good examples of services which must be provided by one or more essential system objects. Such functions certainly do not belong in a minimal closed kernel. Many different policies are possible and the correct choice of policy may depend on the configuration and pattern of use of the system in question. Thus, on the one hand, we have a service which must not be placed in the kernel, and, on the other hand, we have a service which is absolutely essential for the correct operation of our system. Thus the introduction of essential system-objects. By moving such services outside the kernel the minimality of the kernel is retained, as well as the support for extensibility and flexibility is maintained.

System-objects may be viewed as a structured, controlled means of access to kernel functions and data structures. Viewed from a different perspective, system-objects represent a route to the *efficiency* of the kernel. They allow the system-object programmer access to trusted system functions with minimum overhead. System-objects are themselves trusted objects; that is, they are assumed to execute correctly. An error in a system-object may corrupt the entire processor which the system-object is executing on. System-objects provide a traditional object interface to the rest of the system and they are invoked, at the linguistic level, in the same manner as other user objects. Invocations on system-objects are actually implemented by means of protected procedure calls on the kernel for efficiency.

System-objects implement (encapsulate) policy and provide access to kernel efficiency when needed. Most systems provide a set of kernel primitives which can be combined in some fashion to provide new services. However, often the facilities for combining kernel primitives are restrictive and inefficient. For example, if a system-service is implemented by a traditional user-level process each access of that system-service will require an expensive process context switch. System-objects represent a means of providing both extensibility and efficiency while maintaining our

desired goal of a minimal kernel.

System-objects serve as intermediaries for providing system-services. Many such services may be provided by user-level objects and system-objects should be introduced only when efficient access to protected kernel facilities is required. In the spirit of a minimal kernel we have argued that the kernel should contain only those facilities which cannot be located outside the kernel. We make a similar argument concerning system-objects. System-objects should contain only those facilities which cannot be reasonably provided by user-level objects. As an example, a print spooler should probably not be made a system-object. The spooler requires only services which can reasonably be expected to be provided by system-objects such as synchronization and bulk data transfer. A printer device driver, however, is a likely candidate to be made a system object. A device driver will need to access device registers and will need to accept and post interrupts.

System-objects may also be loaded dynamically to introduce and remove services. Thus Ra supports *plug-in* system-objects. There are many advantages to such a feature. First, such dynamic loading allows kernel extensions to be introduced in the form of system-objects without the need for recompiling the kernel and rebooting the system. Second, various configurations of system-services can be loaded based on the capacity and pattern of use of a given machine. For example, a small micro version of Ra/Clouds need not support the full range of system-services provided by larger systems. Third, dynamic loading of system-objects simplifies developing and debugging new system-services. Fourth, multiple versions of a particular service can coexist for comparison and analysis under the same load conditions. For example, two object location mechanisms may be implemented and executed in parallel to compare their efficiency.

Since system-objects may be introduced and removed dynamically, special kernel operations provide a means for facilitating the orderly transfer of responsibility for a given system-service from one system-object to another. Such a transfer of control may, in general, involve a complex protocol or it may, in special cases, be as simple as a change of registry.

System-objects are mapped into the kernel or K space when loaded and remain present, accessible to all processes and objects, until they are unloaded. System-objects are relocatable so that they may be loaded in differing places but this relocation is only done at load time, which is intended to be infrequent.

6.2 Kernel Classes

System objects "see" the kernel through kernel classes. Kernel classes are collections of kernel data and procedures to access and manipulate that kernel data. Thus kernel classes are like

the familiar abstract data types or modules of software engineering. We suggest that they be viewed rather as *mixin classes*. Mixins are special object classes used in various object-oriented programming systems which are designed, not to be instantiated, but rather, to be used to compose new object classes. In this sense, they support a powerful form of multiple inheritance. In Ra, system objects *import* or inherit specified kernel classes. System objects may invoke operations only on kernel classes which have been explicitly imported. Code and data from other kernel classes are protected at the linguistic level¹ and may not be accessed. Run-time protection will probably be too costly in general on traditional hardware although it should be possible to request hardware supported protection while a system object is being debugged.

There are five collections of kernel services. These services are defined as classes. These are the *Virtual Space class*, *Isiba class*, *Synchronization class*, *Device class*, and *System-Object class*.

The virtual space class provides system-objects with functions and data structures that manipulate segments, windows and virtual spaces. Using this class, we can build services that create and delete objects, provide invocation support and so on. The isiba class provide some isiba control primitives. Controlling the computation in the operating system, creating threads and controlling their execution can be achieved through this class, The synchronization class provides isiba synchronization primitives that are useful for locking and recovery control. The device class provides access to the device specific registers and interrupt vectors. These are used by system-objects that implement device services and need to install interrupt handlers. Finally, the system-object class, allow system objects themselves to control the behavior of other system objects, that is installation and repeal of services can be handled.

A more detailed description of the classes and the operations they support is in the appendix.

7 Computation and Storage Spectrums

The various components of the Ra kernel and proposed components of the Clouds operating system may be viewed as being *compositionally* related in the manner displayed in Figure ???. The components above the dotted line represent the three kernel primitive abstractions: segment, isiba, and virtual space. The way in which the kernel primitive abstractions are composed was described in detail in Section 5. Briefly, segments abstract the notion of store and are the most fundamental abstraction. Isibas are constructed from two segments and by abstracting the notion of computation. Virtual spaces are composed by associating segments with ranges of virtual addresses and then storing the information describing the association in yet another

¹That is, at compile time.

segment. Virtual spaces abstract the notion of addressing domain and provide a means for manipulating virtual memory page and segment tables.

Proposed entities appear above the dotted line and are composed of the Ra primitive abstractions. These entities may be viewed abstractly as falling within one of two *spectrums*, either the computation or storage spectrum. Entities within the storage spectrum represent successive *specializations* of the abstract notion of store. Segments, as we have said, are the most primitive. Virtual spaces are composed of segments. Clouds objects are composed of a structured virtual space and an invocation mechanism. Objects can be further refined to produce recoverable objects[WL86] by placing additional restrictions on the behavior of object storage. One can imagine the spectrum continuing with further specializations of recoverable objects. Similarly, variations of entities within the spectrum are possible. Given the virtual space primitive abstraction, various forms of object can be composed by using alternate invocation mechanisms or structure.

Parallel to the storage spectrum is the computation spectrum. Isibas represent the most basic form of activity. Clouds processes are then composed of one or more isibas in conjunction with a virtual space. Next appears the Clouds thread. Threads are composed of a possibly distributed collection of processes. The computation spectrum can also be viewed as a specialization hierarchy. Actions[WL86] are threads which make certain guarantees about the recoverability and atomicity of computations which run on their behalf. As with the storage spectrum, variants of the basic entities can be proposed and may exist concurrently within the system.

Viewing entities in Clouds/Ra as existing on these spectrums provides a uniformity and structure to our system and appears as a natural result of Ra's extensibility. System objects and kernel classes are the mechanisms by which entities are composed. The storage and computation spectrums also provide a structured approach for exploring the design space of operating system components.

8 Using the Primitives

In this section we describe the implementation of some of the basic functionality of the Clouds operating system using Ra's primitive abstractions.

8.1 Implementing Clouds

An overview of the Clouds system was given in Section 2. Here we show how the different Clouds components can be built using the Ra toolkit of primitives.

8.1.1 Objects

Clouds objects are structured Ra virtual spaces which are mapped into the O-space when invoked. The segments constituting an object are distinguished by various attributes which are based roughly on whether the segment contains code or data. For example, code segments are read-only, while a segment holding recoverable data has a recovery attribute which enables the kernel to locate the partition responsible for handling page faults on the segment.

Objects are invocable which provides a structured means for an execution to enter and leave the virtual space associated with the object. Object invocations may be nested and thus consist of a call which transfers control to the specified operation in the desired object, and a return which transfers control back to the original calling object. A system object called the *invocation manager* is responsible for installing the appropriate object virtual space on object call and return. Consider the following typical scenario: an isiba executing in a user object invokes an operation on another object. A user-level object invocation appears to the Ra kernel as a request on a system object, the invocation manager. The kernel protected procedure call mechanism verifies that the specified system object, the invocation manager in this case, appears in the system object directory. The invocation system object is then called with four parameters: the invoking object, the object to be invoked, and a segment containing the object-call parameters, and the desired operation. When an object is found locally, the invocation manager installs the object's virtual space, and "returns" by transferring control to the location in the newly installed object virtual space corresponding to the desired operation. A similar scenario is followed on object return. (Section 8.1.2).

If the object to be invoked is not found on the local node, the invocation manager may solicit the help of other system objects to locate the desired object. This may be done by using the partition manager as described in Section 5.4 to activate the segment which contains the object's descriptor, or it may involve a more complex object location policy² performed by an *object locator* system object. Both the partition manager and the object locator would typically perform their function by communicating with their counterparts on remote nodes. Once the desired object has been identified on a remote node, the object is brought (whole or partial) from the remote node, installed in the O space and execution is resumed. The partitions in charge of mediating accesses to the invoked object segments maintain the coherency of these segments among the various nodes using them. The coherency scheme is encoded in the partitions, and different coherency schemes can be used to maintain the consistency of different segments.

²Such as one of those described in [Ber87].

8.1.2 Threads

Threads are the only visible active entities in Clouds. A thread is a unit of activity from the user's perspective. Upon creation, a thread starts executing in an object by invoking an entry point in the object. Threads may span machine boundaries.

A thread is implemented by one or more processes. A process is a virtual space with one isiba associated with it, plus other process-specific information – data that is logically associated with the activity rather than with an object, such as controlling terminal, thread specific information, and action specific information. The process contains *stack segments*, one for each object invocation. These segments are created and destroyed in a LIFO manner. A process is created for each thread's visit to a machine.

Process management is handled by the *process manager* system object. A thread is created as follows: the process manager object creates a process virtual space that include a stack segment plus other thread related information. Next, it creates a segment and initializes it as an isiba context segment (note that only the machine independent part is initialized; the kernel manages the machine dependent part). The process manager sets the scheduling attributes of the isiba, sets the stack segment in the process virtual space as the isiba stack, and initializes the P space attribute to the process virtual space and the O space attribute to the object virtual space to be invoked.³ The address to start execution is included in the initialization of the machine.independent part of the control segment. Lastly, the process manager calls the isiba scheduler to make this isiba runnable.

8.1.3 Synchronization

The *synchronization manger* is responsible for intra-object synchronization. Each object contains one or more *lock* segments. A lock segment is a read-write segment (which is protected for system-mode access only). It contains lock information plus associated queues that indicate which threads (and isibas) are waiting on a lock, if any. A thread running in an object requests a lock operation as follows: the thread performs a kernel call giving the synchronization manager as a parameter, plus lock and mode requested. The kernel routes the call to the synchronization manager. The synchronization manager examines the appropriate lock segment in the object, and determines whether or not to grant the lock request. If it decides that the isiba has to block waiting on the lock to be released, it makes a note of this fact in the lock segment and calls the isiba scheduler to block the thread. When a threads releases a lock, the synchronization

³A thread starts its life by invoking an object. The O space attribute of the isiba is set to point to the virtual space of this object.

manager inspects the lock segment. If there are any waiting threads which should be granted the lock, a note is made in the lock segment, and the isiba scheduler is called to schedule the isibas unblocked by the lock release operation.

The exact semantics of the locking schemes used are specified by the synchronization manager and not encoded in the kernel. Different synchronization primitives can be implemented and experimented with by modifying the synchronization manager. By separating the *mechanisms* of implementing synchronization variables from *policies*, by placing the actual synchronization queues and data in the object, and by entrusting system objects with the task of implementing the policies, we achieve a flexible synchronization mechanism which we believe will be easier to implement, maintain, and modify.

9 Conclusions

The Ra kernel is a minimal kernel for the Clouds operating system. The Ra design is modular and Ra supports extensibility. Ra uses the object-based paradigm supported by Clouds to implement Clouds, providing the first instance of an application of the paradigm.

The implementation of Ra is under progress. Ra is being implemented on a network of Sun-3/60 workstations. The workstations will run Ra, and the partitions will reside on a file server running Unix. The partition code will run as a Unix application program.

All user application code and user services will be implemented as a part of the Clouds system, and will run on top of Ra. User interfaces are not designed yet, but preliminary plans are to use X-windows and a Unix like shell for user interactions. Another scheme under consideration will use Unix as a front end user interface that allows access to Clouds. In this scheme the user workstation will run Unix, a set of rack mounted compute servers will run Clouds and a set of file servers will host the partitions, which can run under Unix, or in native mode under Ra.

References

- [AAJK87] Mustaque Ahamad, Mostafa H. Ammar, José M. Bernabéu, and M. Yousef A. Khalidi. *Locating Resources in a Distributed System Using Multicast Communication*. Technical Report GIT-ICS-87/44, Georgia Institute of Technology, December 1987.
- [ABA88] Mostafa H. Ammar, José M. Bernabéu Aubán, and Mustaque Ahamad. Using hint tables to locate resources in distributed systems. In *IEEE INFOCOM'88*, 1988.

- [ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: a technical review. *IEEE Trans. on Software Eng.*, SE-11(1):43–58, Jan. 1985.
- [Ber87] José M. Bernabéu Aubán. *Locating Resources in Distributed Systems*. PhD thesis, School of Information and Computer Sciences, 1987. PhD Research Proposal.
- [CD87] Raymond Chen and Partha Dasgupta. *Consistency Preserving Threads: An Approach to Atomic Programming*. Technical Report GIT-ICS-87/43, School of ICS, Georgia Institute of Technology, December 1987.
- [CZ83] D. R. Cheriton and W. Zwaenepoel. The distributed V kernel and its performance for diskless workstations. *Operating Systems Review*, 17(5):128–140, October 1983.
- [DLA88] P. Dasgupta, R. J. LeBlanc, and William F. Appelbe. The Clouds distributed operating system: functional description, implementation details and related work. In *Proc. of the 8th Intl. Conference on Distributed Computing Systems*, IEEE, June 1988.
- [Gif79] D. Gifford. Weighted voting for replicated data. In *Proceedings of 7th Symposium on Operating Systems (Pacific Grove, California)*, ACM, Dec 1979.
- [HMSC87] Roger Haskin, Y. Malachi, Wayne Sawdon, and Greg Chan. Recovery management in Quicksilver (extended abstract). In *Proc. Eleventh ACM Symp. on Operating Systems Principles*, pages 107–108, November 1987.
- [J*85] E. D. Jensen et al. *Decentralized System Control*. Technical Report RADC-TR-85-199, Carnegie Mellon University and RADC, Pittsburgh, PA, April 1985.
- [LH86] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proc. 5th ACM Symp. Principles of Distributed Computing*, pages 229–239, ACM, August 1986.
- [Lis83] Barbara H. Liskov. Guardians and actions: linguistic support for robust distributed programs. *TOPLAS*, 381–404, July 1983.
- [Mye82] Glenford J. Myers. *Advances in Computer Architecture*. John Wiley, 1982.
- [Pit86] D. V. Pitts. *Storage Management for a Reliable Decentralized Operating System*. PhD thesis, School of Information and Computer Science, Georgia Tech, Atlanta, Ga, 1986. (Technical Report GIT-ICS-86/21).

- [RAK87] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. *Hardware Support for Distributed Shared Memory*. Technical Report GIT-ICS-87/41, School of ICS, Georgia Institute of Technology, November 1987.
- [RR81] R. F. Rashid and G. G. Robertson. Accent: a communication oriented network operating system kernel. In *Proc. of the Eighth Symposium on Operating Systems Principles*, pages 64–75, December 1981.
- [Spa86] E. H. Spafford. *Kernel Structures for a Distributed Operating System*. PhD thesis, School of Information and Computer Science, Georgia Tech, Atlanta, Ga, 1986. (Technical Report GIT-ICS-86/16).
- [STB86] R. E. Schantz, R. H. Thomas, and G. Bono. The architecture of the Cronus distributed operating system. In *Proc. of the 6th Int'l. Conf. on Distr. Computing Sys.*, May 1986.
- [TM81] A. S. Tanenbaum and S. J. Mullender. An overview of the Amoeba distributed operating system. *Operating System Review*, 13(3):51–64, July 1981.
- [WCC*74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Commun. of the ACM*, 17(6):337–345, June 1974.
- [WL86] C. T. Wilkes and R. J. LeBlanc. Rationale for the design of Aeolus: a systems programming language for an action/object system. In *IEEE Computer Society 1986 International Conference on Computer Languages*, 1986.

A Kernel Classes

The kernel classes are introduced in section 6.2. The following sections outlines the operations associated with each class.

A.1 Virtual Space Class

Activate Attach the segment containing a description of the specified virtual space (the virtual space descriptor) into K space and prepare to map the virtual space into the P or O space. Allocate segment and page mapping structures.

Deactivate Deallocate segment and page mapping structures. Detach the virtual space descriptor from K space.

Install Map the specified virtual space into P or O space. Entails constructing the appropriate segment and page tables and, possibly, replacing the currently installed virtual space. May require activating the virtual space.

Attach Segment Form an association between a specified segment and a range of addresses in a specified virtual space. Update the virtual space descriptor to reflect this association. May be performed either dynamically when the specified virtual space is either mapped in P or O space, or passively when the virtual space is not mapped. The specified segment may be either activated or deactivated.

Detach Segment Dissociate the specified segment with the range of addresses in the specified virtual space. Note that a single segment may be mapped into a virtual space more than once so that detaching a segment from one location may not completely remove the association of the segment and virtual space. May be performed either dynamically or passively.

Query-Update Read or modify current values in virtual space descriptor.

A.2 Isiba Class

Install Request that the specified isiba be assigned a physical processor. If none are currently available, place the isiba on the ready queue. A time quantum, a maximum execution time, and a priority level are specified with this call. The ready list is maintained as a priority queue. The isiba must have associated P and O space attributes, a scheduler attribute, and a privilege level. The isiba will receive quantum units of processor time until the maximum execution

time is reached or until it is removed. The associated P and O spaces may be installed repeatedly as a result of this operation.

Remove Remove the specified isiba from the ready list and place it at the specified destination (typically a queue associated with a synchronization variable), or, if the operation is requested by the target isiba itself, remove the isiba from the processor and dispatch a new isiba. The associated P and O spaces may remain mapped if the next isiba to be dispatched has the same attributes (that is, if more than one isiba is executing in the specified virtual spaces).

Query-Update Read or modify current values in isiba context segment or stack segment.

A.3 Synchronization Class

These operations are intended to be used only for synchronizing system objects. See section 8.1.3 for a description of how inter-object synchronization can be achieved in Ra.

Acquire Request control of synchronization variable. Parameters include type of synchronization variable, instance of type, and action desired (such as 'block if unavailable' or 'return immediately'). Examples of possible synchronization variable types include counting semaphores, spin locks, read-write locks, events, etc.

Release Return previously acquired control of a synchronization variable. Parameters are as for the acquire operation. May cause blocked activities to awaiting the synchronization variable to be resumed.

A.4 Device Class

Mount Associate a controlling system object (device driver) with a given piece of hardware. Involves installing the appropriate interrupt handlers and initializing the device.

Unmount Bring the specified device to quiescence and replace the associated interrupt handlers with null handlers.

Query-Update Read or modify current values in the device registers, interrupt vectors, or other device-related data structures.

A.5 System Object Class

Load This operation causes a window of the virtual space installed in K space to be mapped to the segment containing the system object code. After the mapping is done, it will usually be necessary to link the system object's code to the rest of the kernel, performing any needed relocation of the object's code. After the above is done, the specified system object is registered in the system object directory thereby making it available for user-level invocations. Register the system object's dispatch table thereby making it available for inter-system object invocations. The registration step may involve the replacement of a previous system object which was already performing the function the new object is to perform. When this is the case, it will be necessary to provide a way of transferring information from the old object to the new one, before the old object is unloaded.

Unload Remove the specified system object from the system object directory and remove its dispatch table. Deallocate the memory occupied by the system object. The unload operation will be performed only if the object being unloaded is not the only one containing information which is vital for the orderly operation of the system.

Query-Update Read or modify current values in the system object directory (used for user-level invocations of system objects) or the system object dispatch tables (used for inter-system object invocations).

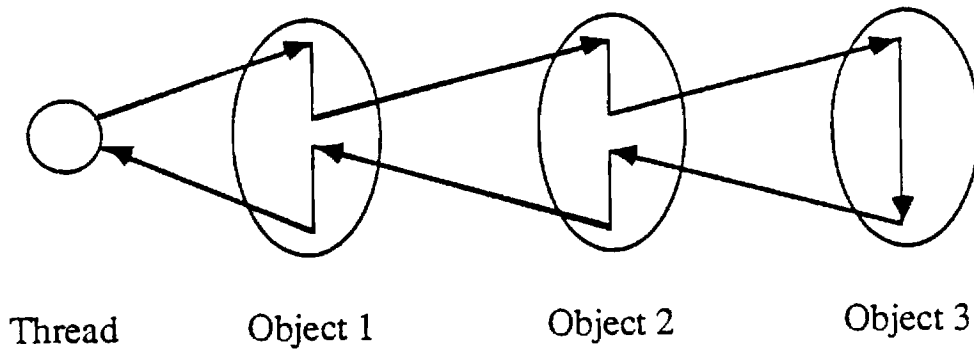


Figure 1: Clouds invocation model

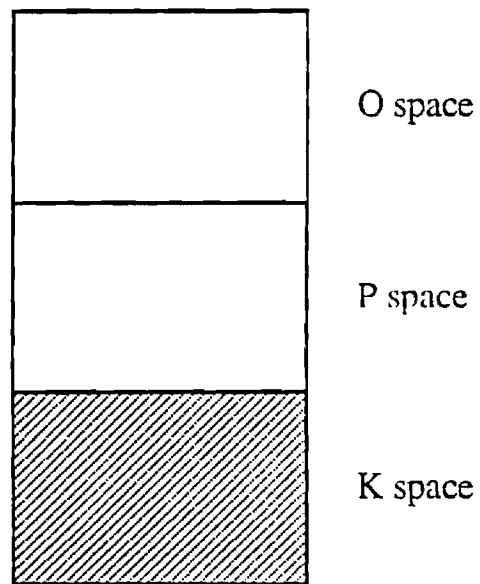


Figure 2: Hardware virtual memory structure

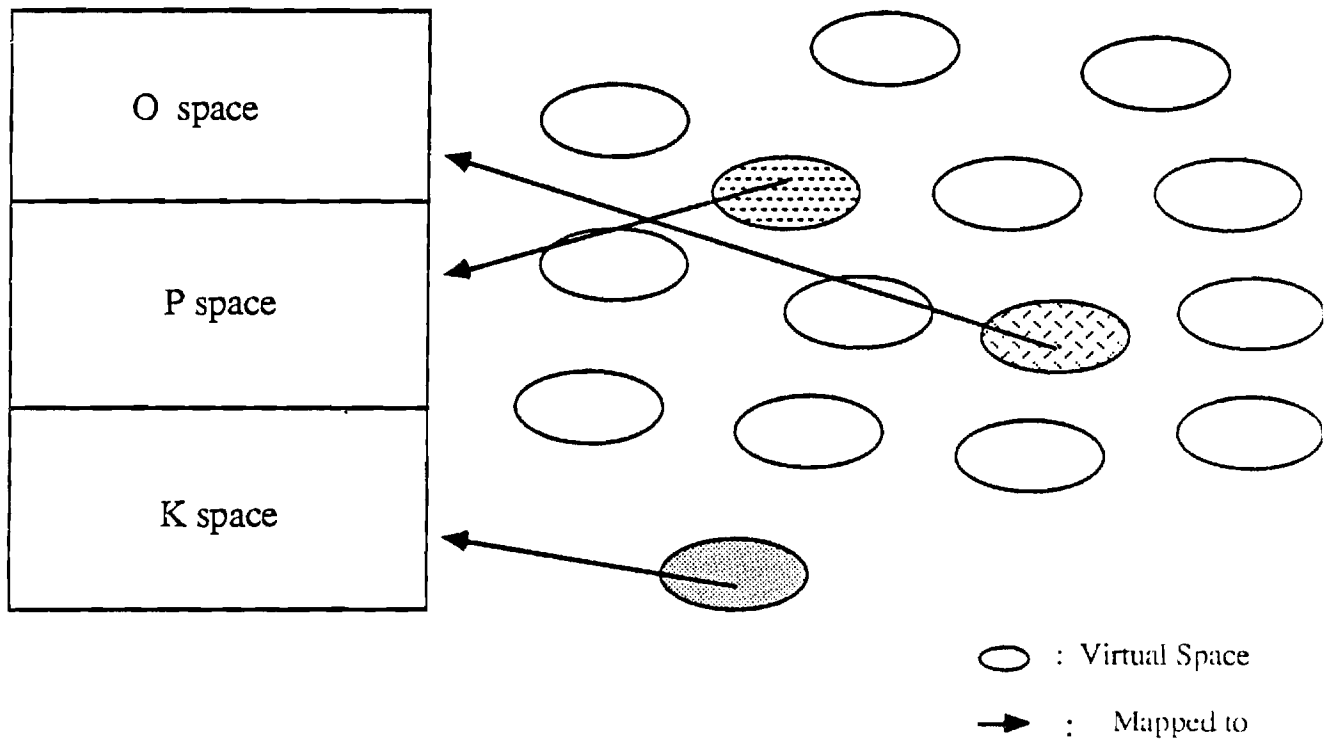


Figure 3: Mapping virtual spaces into hardware spaces

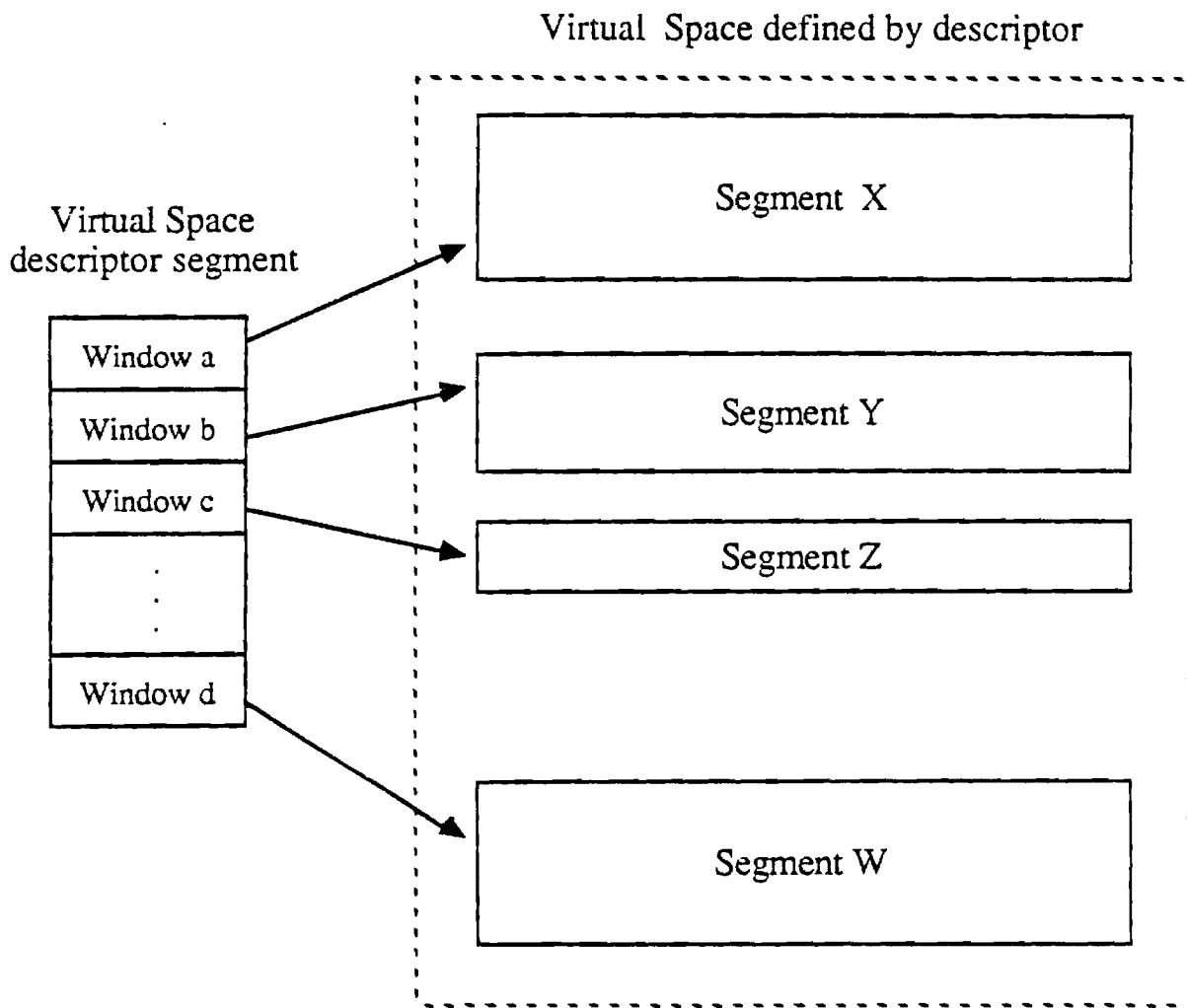


Figure 4: Segment describing a virtual space

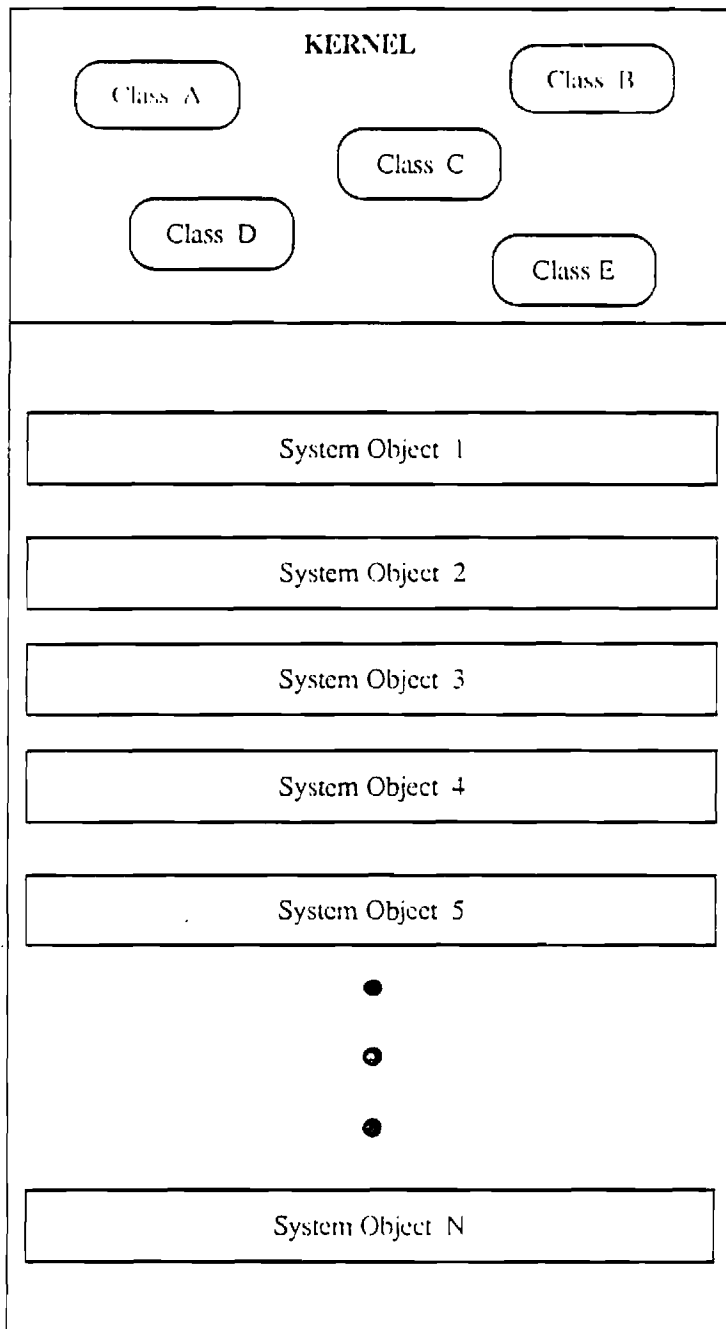


Figure 5: Kernel virtual space

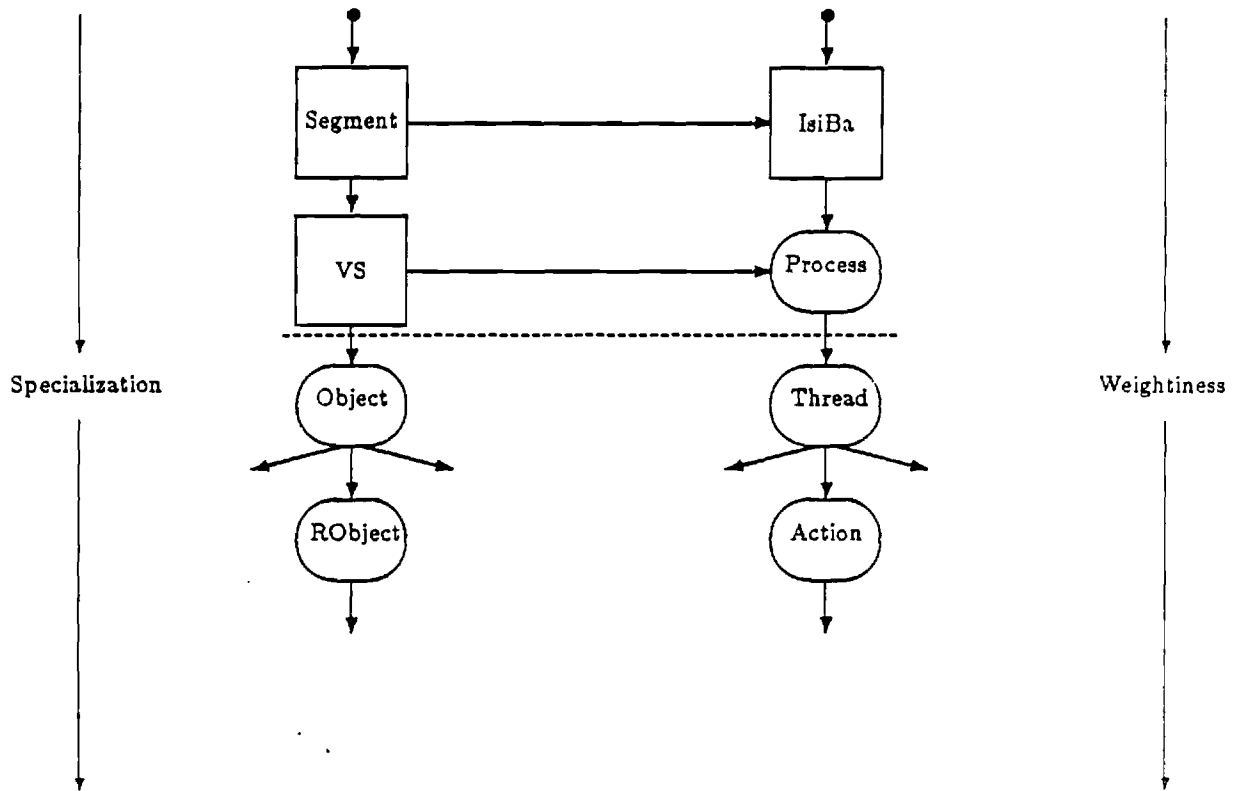


Figure 6: Ra Primitive Composition Spectrums

A Support Architecture for
Reliable Distributed Computing Systems

Semi-Annual Technical Report
July 1986 to December 1986

From:

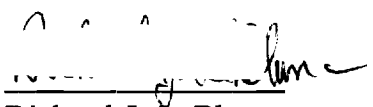
Georgia Tech Research Corporation
Atlanta, Georgia 30332

To:

National Aeronautics and Space Administration
Langley Research Center

Grant No.: NAG-1-430

Endorsements:



Richard J. LeBlanc
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2592



Partha Dasgupta
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572

A Support Architecture for Reliable Distributed Systems

1. Introduction

The Clouds Project at Georgia Tech is conducting research aimed at building a reliable distributed operating system. This document provides an overview of the goals of the project, the significance of the funding from NASA towards these goals, the progress of the project over the last year, the current status and future plans.

The *Clouds* project is funded currently by NASA and by RADC. The project was initially funded in part by NSF, ONR and NASA. The current funding has enabled us to make substantial progress and achieve recognition in the research community as a significant effort in research in distributed systems.

2. NASA Contribution

The grant from NASA has supported over the last year two graduate students, one month of faculty time, some secretarial and supplies. This has been of significance help for the project to acquire its present status. Although the support has not been enough to provide adequate support for a project of the size of *Clouds*, along with some support from RADC and internal funds from Georgia Tech, has gone a long way towards our progress.

The NASA support has has two significant outcomes, the first leading to the second.

- 1] The funds were used to support two graduate student who produced the first usable version of the *Clouds* kernel. The kernel currently runs as a distributed system and supports most of the basic *Clouds* functions.

- 2] The first version of the kernel coupled with research results obtained thus far, and increased interest in *Clouds* led us to produce a NSF-CER proposal for major funding of the infrastructure of distributed systems development with *Clouds* as a central theme. This proposal has had significant positive reaction from NSF and stands in good chance of being funded. The NASA support served as an excellent source of seed funding for our long term plans.

3. Research Goals

The research goals of *Clouds* is to design, implement and test an approach to constructing distributed systems that has the promise of being the architecture of the next generation of systems. Some of the following paragraphs are summarized from previous reports and the reader is referred to them for a more complete discussion of the *Clouds* design and implementation strategy. The following are the requirements of our distributed system.

- 1] The operating systems will be distributed over several sites. The sites will have a fair degree of autonomy. Yet the distributed system should work as an integrated system. Thus the system should support location independency for data, users and processes.
- 2] Reliability is a key requirement. Large distributed systems use significant number of hardware components and communication interfaces, all of which are prone to failures. The system should be able to function normally even with several failed components.
- 3] Distributed systems often have dynamic configurations. That is, newer hardware gets added, or faulty hardware is removed. The system function should not be hampered by such maintenance chores. Thus the system should be dynamically reconfigurable.

- 4] The system should be capable of monitoring itself. This encompasses hardware monitoring for keeping track of hardware failures as well as monitoring key software resources (for example daemons, network servers, and so on.) On detection of failure the system should be able to self-heal (restart daemons) or self-reconfigure (eliminate faulty sites).
- 5] The users should be shielded from both the configuration of the system (site independence) as well as its failure modes. For example, if the site a user is connected to fails, he should be transferred to an active site transparently.
- 7] Many of the above functions can be implemented on conventional systems, but would make the system extremely slow. Thus efficiency is an important design criteria.

The above requirements has been designed into the Clouds operating system. Most of the functions have been designed into the kernel of the system, without making the kernel too complex, bulky or inefficient. The design philosophies adopted for the Clouds operating system are:

4. Implementation Strategy

The following items describe in brief the general approaches taken to implement the goals of the system.

- 1] An object-based, passive system, paradigm is used as the basic architecture. All system functions, data, user programs and resources are encapsulated as passive objects. The objects can be invoked at appropriate entry points by processes.
- 2] The objects in Clouds represent nearly everything the system has to offer. The site independence philosophy is implemented by making the object name space (system names) flat and site independent. When a process on any machine invokes an object

located anywhere, no site names are used. Hence the location of any particular object is unknown to a process.

- 3] Reliability is achieved through two techniques. One of them is the action and recovery concept. The action mechanisms are supported at the kernel level. Actions are atomic units of work. Any unfinished or failed action is recovered and has no effect until it completes.

Reliability is augmented from just consistency preservice to (limited) guarantees of forward progress through fault tolerance mechanisms. These include replication of resources and objects for higher availability and replication of computation for failure masking.

5. Future Research Plans

The following projects are ongoing as our effort to build the *Clouds* distributed system continues. These projects are not all funded and are partially being supported by the funds from NASA. The aim of the research is to build a usable, useful and reliable distributed computing environment.

5.1. Reliable Object Filing System

The persistent object scheme used in *Clouds* resides in a flat name space. This project is researching schemes to create a structured name space that is global, and yet not prone to failures. A tree-like hierarchy of names can be easily obtained through directory objects, but has the serious problem that if the root is unaccessible, the entire system is unusable.

A reliable object system can be implemented through capability caches and replication at several sites of key directory objects and using the distributed hint-base for creating the hierarchical structure. The algorithms to maintain availability as well as

consistency id under investigation.

5.2. Replication and the PET Scheme

Replication and Parallel Execution Threads (PET) are a powerful combination of data as well as computation redundancy that implements fault tolerance in software. This scheme is a generalized version of replication used in the Circus system and in the ISIS system. Preliminary details of the scheme has been presented at the 6th Symposium on Reliability in Distributed Software and Database Systems and an expanded paper has been submitted to the IEEE Trans. on Software Engineering.

The project dealing with fault tolerance is looking at completing the design of the PET scheme and study effective and efficient mechanisms to implement it in the *Clouds* operating system.

5.3. Probe System for Monitoring

One of the key functions of managing a distributed environment is to be able to monitor the state of the system, identify failures and reconfigure the system on demand or need. The probe system is a mechanism that effectively allows these functions to be implemented. The algorithms and the mechanisms used have been presented in the First OOPSLA conference. This project is looking at the full scale design of the monitoring system and its implementation

5.4. Object Locator System

Clouds is a system with no global state information, and completely transparent object storage. As a result of this objects have to be located by searching the entire distributed system. Some hint information is stored and last accesses are cached providing some significant performance gains through short circuiting the searches. However global searches may still be necessary, but can get expensive.

This project is designing mechanisms that lower the object location overhead significantly. One of the preliminary results have shown that binding encrypted object capabilities to multicast addresses result is a very large improvement in predicted performance. Other approaches with higher payoffs and easier implementations are being studied.

5.5. Object Programmer's Toolkit

The world of objects created by *Clouds* is not a familiar world for most programmers. To get users used to the *Clouds* environment we are implementing an object programmers toolkit consisting of various tools including Unix style files, streams, pipes, process control primitives as well as debuggers, library functions, system services, and so on. This is a part of a larger project aimed at creating a friendly distributed environment that mends a major reason for the lack of popularity of distributed systems.

6. Progress

As of present we have an operational distributed kernel running on a set of VAX-11/750 computers. The *Clouds* kernel has been debugged and is running as an experimental release on two VAX-11/750's.

The kernel supports persistent objects, object invocations (both local as well as remote), rudimentary name services and other standard functions such as virtual memory, storage management, synchronization and scheduling. The Aeolus compiler is operational and is being used for object compilation for both systems level as well as user level programming for *Clouds*.

On the short term the kernel development is continuing in the areas of action management (the transaction system, that will be available soon), the user interfaces and preliminary toolkit functions, object naming services, additional disk drivers and fixing bug reports. The long term research and development is progressing as described above.

7. Conclusions

The *Clouds* project has been progressing well and the NASA funding has allowed us to make significant strides towards the goals of the progress. We feel further funding in this area is extremely necessary for continued growth of research into distributing computing systems.

**A Support Architecture for
Reliable Distributed Computing Systems**

**Semi-Annual (interim) Technical Report
January 1987 to June 1987**

From:

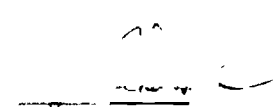
**Georgia Tech Research Corporation
Atlanta, Georgia 30332**

To:

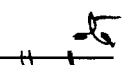
**National Aeronautics and Space Administration
Langley Research Center**

Grant No.: NAG-1-430

Endorsements:



**Richard J. LeBlanc
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2592**



**Partha Dasgupta
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572**

A Support Architecture for Reliable Distributed Systems

1. Introduction

The Clouds Project at Georgia Tech is conducting research aimed at building a reliable distributed operating system. This document provides an overview of the goals of the project, the significance of the funding from NASA towards these goals, the progress of the project over the preceding 6 months since the inception of the 1987-1988 funding.

The *Clouds* project is funded currently by NASA and by RADC. The project has just received funding under the National Science Foundation's Co-ordinated Experimental Research Program (known as the NSF-CER program) for the development of a distributed computing environment and for support of research in various fields impacted by distributed computing.

2. NASA Contribution

The NASA funding received so far for the *Clouds* project has been a major source of seed funding which allowed us to obtain NSF funding. The NSF funding has not reached us yet. The current effort is being funded largely by NASA. This report outlines the significant steps taken by the *Clouds* project in the last 6 months and outlines our plans for the next 6 months. Continued research beyond that time frame will be dependent on continuing support from NASA.

3. The Clouds Kernel: Towards version 2.

The *Clouds* kernel described in the previous reports is the version 1 kernel. This kernel is complete, has been tested and is frozen. No further research except for some application development work is underway with this version of the kernel.

The v1 kernel has some flaws which became apparent as usage of the kernel became more widespread. The kernel design has some shortcomings. The monolithic nature of the implementation is making adding functionality difficult and the number of bug fixes necessary is not showing sign of stabilizing.

Thus we have taken the decision to start work on the next version of the *Clouds* kernel. This kernel, internally called “*Ra*” will be a completely redesigned and re-implemented kernel for the *Clouds* operating system, which supports the *Clouds* paradigm as described in earlier reports. *Ra* will be designed to a new set of guidelines.

Ra is the first object-oriented kernel of an object based distributed system. The kernel not only supports *Clouds*, but is inherently object oriented. Some of the salient features of *Ra* are:

- 1] The *Ra* kernel is small, lightweight and stripped of all possible system management duties. *Ra* supports object invocations and scheduling. No other functions are supported in the *Ra* kernel. This is in line with the idea that operating systems should be supported by a “minimal” kernel, and all other operating system functions should be supported by service modules that sit on the kernel. The advantages of this approach include (but not limited to) faster implementations, less debugging, modularity, easy upgrades, extensibility and elegance.
- 2] The system functions that were part of the *Clouds* v1. will be supported by “system objects”. These objects will include modules such as memory management, object mapping, communications, storage drivers, storage manager, action management and so on.
- 3] The systems objects will be dynamically linkable to *Ra*, allowing changes in the system support at runtime.

4. Design and Development

The design of Ra is under progress, and includes the cooperative effort of 3 faculty and 3 graduate students. 3 other faculty and about 4 graduate students are serving as the advisory committee overseeing the design decisions and providing the feedback as to the appropriateness of the functionality and mode of implementation.

The design is about midway. We have designed a segment based object management system, identified the kernel functions, and are developing the concurrency and recovery handling strategies. Also the system interface specification is being developed.

4.1. Milestones

The first phase of the design will be completed in September. We expect the design to go through some revisions and be ready by November. The implementation of the basic kernel facilities will start in October. This will include the basics such as interrupt handling, kernel data structures, scheduling and virtual memory.

After the final design of Ra and allied system object are ratified, the major implementation work will commence. All the systems objects and the kernel will be implemented. The design will stress to make the implementation scheme machine independent. The first implementation will be targeted to the AT&T 3B2 series of computers. These offer a very sophisticated set of hardware features that are well suited for kernel development (this includes, segmented memory management, communication co-processors, and in-depth documentation of hardware internals).

We expect a rudimentary version of *Clouds* v2.0 (with the Ra kernel) to be operational in early 1988 (Jan-Feb). From that point onwards both the system development as well as distributed systems research using the *Clouds* testbed will proceed in parallel.

5. Applications Development

Along with the design and development of the Ra kernel, we are continuing our efforts of developing application environments and system facilities on top of *Clouds* v1.

5.1. Pandora

Pandora is the object programmers toolkit. Pandora is built as a set of *Clouds* object that provide a variety of services and create a friendly distributed environment for *Clouds* systems programmers. Pandora v0 has been implemented and consists of filing services, concurrent programming support (pipes, message) and some shared data structures.

Research is underway to identify the proper tools needed by system programmers and develop the designs and implementations of these. The current topics under study include object binding schemes, user services, debugging support, directory service and so on.

6. Conclusions

The *Clouds* project has benefited widely through support from NASA Langley. The support is currently being used in the development of Ra, allied systems support and peripherally in the development of Pandora.

The longer term benefits of this research will include:

- 1] Development of a paradigm for distributed computing that will enhance the usability of distributed systems in all forms of applications from general purpose systems to embedded systems.
- 2] Discover and develop technology for implementing efficient, reliable and fault tolerant distributed systems.

- 3] Make distributed system move from the experimental domain and enter general purpose use. This can be achieved through our usage of a appealing paradigm, provision of sufficient application oriented tools and building a system that performs efficiently.
- 4] More usage of distributed system will have a large payoff, that is will cause a wider acceptance of these systems leading to faster applications development, which will lead to substantial strides in the technology of building distributed systems.

6.1. Future Directions

We believe that distributed systems research will greatly benefit the computing community. We are addressing all the concerns that we can conceive currently. The last 6 months was an exciting period for *Clouds* due to the rapid deployment of newer technology, promise of further support from NSF, and the successful start of the design of Ra. Our next proposal to NASA will include a better overview of Ra and the structure of the new *Clouds* v2. It will also contain details of our upcoming research thrusts which includes the following areas:

- 1] The design and fabrication of VLSI based MMU structures for efficient support of object based systems. *Clouds* requires, for efficiency, an advanced memory management system, not supported by any commercial chip. We plan on developing a prototype.
- 2] The implementation of a highly fault tolerant system. Using object level replication and a scheme called "PET" we can achieve failure transparency for application.
- 3] The next generation of language support for reliable object oriented systems. These languages will support apart from the functionality supported by *Aeolus*, facilities such as inheritance, reliable invocations, failure transparency etc.

- 4] Effective resource management in distributed systems. Strategies to effectively manage all the resources, resolve distributed naming and provide global integrated systems services.
- 5] Object programmers toolkit. This will lead to the later versions of Pandora.

6-10-85

INTERIM TECHNICAL REPORT

**A SUPPORT ARCHITECTURE FOR RELIABLE
DISTRIBUTED COMPUTING SYSTEMS**

Prepared for

National Aeronautics and Space Administration
Langley Research Center

Under

Grant No. NAG-1-430

Final Report for Period November 9, 1983 to December 3, 1985

GEORGIA INSTITUTE OF TECHNOLOGY
A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA
SCHOOL OF INFORMATION AND COMPUTER SCIENCE
ATLANTA, GEORGIA 30332

1985



A Support Architecture for
Reliable Distributed Computing Systems

Interim Technical Report
November 9, 1983 - December 3 1985

From:

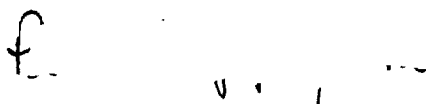
Georgia Tech Research Corporation
Atlanta, Georgia 30332

To:

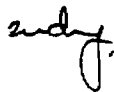
National Aeronautics and Space Administration
Langley Research Center

Grant No.: NAG-1-430

Endorsements:



Martin S. McKendry
Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572



A Support Architecture for Reliable Distributed Systems

1. Introduction

The Clouds Project at Georgia Tech is conducting research aimed at building a reliable distributed operating system. The primary objectives of the Clouds operating system are:

- 1] The operating systems will be distributed over several sites. The sites will have a fair degree of autonomy. Yet the distributed system should work as an integrated system. Thus the system should support location independency for data, users and processes.
- 2] Reliability is a key requirement. Large distributed systems use significant number of hardware components and communication interfaces, all of which are prone to failures. The system should be able to function normally even with several failed components.
- 3] The processing environment should guard against both hardware and software failures. The permanent data stored in the system should be consistent.
- 4] Distributed systems often have dynamic configurations. That is, newer hardware gets added, or faulty hardware is removed. The system function should not be hampered by such maintainance chores. Thus the system should be dynamically reconfigurable.
- 5] The system should be capable of monitoring itself. This encompasses hardware monitoring for keeping track of hardware failures as well as monitoring key software resources (for example daemons, network servers, and so on.) On detection of failure the system should be able to self-heal (restart daemons) or self-reconfigure (eliminate faulty sites).
- 6] The users should be shielded from both the configuration of the system (site independence) as well as its failure modes. For example, if the site a user is connected to fails, he should be transferred to an active site transparently.
- 7] Many of the above functions can be implemented on conventional systems, but would make the system extremely slow. Thus efficiency is an important design criteria.

The above requirements can be handled by a distributed system and have been designed into the Clouds operating system. Most of the functions have been designed into the kernel of the system, without making the kernel too complex, bulky or inefficient. The design philosophies adopted for the Clouds operating system are:

- 1] An object-based, passive system, paradigm is used as the basic architecture. All system functions, data, user programs and resources are encapsulated as passive objects. The objects can be invoked at appropriate entry points by processes.
- 2] The objects in Clouds represent nearly everything the system has to offer. The site independence philosophy is implemented by making the object name space (system names) flat and site independent. When a process on any machine invokes an object located anywhere, no site names are used. Hence the location of any particular object is unknown to a process.
- 3] Reliability is achieved through two techniques. One of them is the action and recovery concept. The action mechanisms are supported at the kernel level. Actions are atomic units of work. Any unfinished or failed action is recovered and has no effect until it completes. The recovery mechanisms are supported inside every object an action touches.
- 4] Reliability is further extended by the self monitoring and self reconfiguration subsystems. This is a set of monitoring processes that use "probes" to keep track of all key system resources, both hardware and software. On detection of failed or flaky components, the monitoring system invokes the reconfiguration system which rectifies or eliminates (if possible) the faulty components, and initiates recovery of affected actions. The monitoring and reconfiguration subsystems are also monitored by the monitoring system.
- 5] The consistency requirements of the data are handled by the recovery mechanisms and by concurrency control techniques. The concurrency control is handled by synchronization paradigms that are an integral part of the object handling primitives. The synchronization of processes executing in an object is handled automatically by semaphores that are a part of the object. This gives rise to a two-phase locking algorithm that is supported by the kernel as a default. The object programmer has the choice of overriding these controls and use custom built concurrency control, depending upon the application. It is also possible to turn off the default recovery and commit strategies.
- 6] Efficiency has been of concern. The object invocation, recovery and synchronization are handled by the kernel. It turns out that these can be done at the kernel level without much overhead. Since the entire Clouds design is primarily based on object manipulations, invocation and synchronization will be the most used operations. Implementing them at the kernel level will result in an efficient system.
- 7] The site independence at the user level is handled in part by using intelligent terminals. The user terminals are not hard-wired into any machine or site, but are on an ethernet, accessible by any site. Each user session is, of course, handled by one

particular site, but any failure causing the controlling site to be unaccessible causes the user to be transferred to another site. This is handled cooperatively by the user terminal and the other sites. Thus the user terminals are actually intelligent microprocessor systems on the Clouds ethernet. In addition to cooperation with the Clouds network, the user terminals run "Bubbles", a multiwindowing, user-friendly interface to Clouds.

2. Progress Report

The following is a brief report of the current status of the implementation of Clouds.

2.1. Equipment

The test equipment for implementing Clouds was funded by the National Science Foundation and has arrived. Three VAX/750 computers interconnected by a ethernet was installed in November 1984. They have been heavily used to develop the Clouds kernel and allied software described later. Three IBM-PC/XT computers, arrived in July 1985 and are being used to develop the intelligent terminal interface to Clouds. One IBM-PC/AT is scheduled to arrive soon and will be used as the primary development system for Bubbles and the ethernet handling code for the terminal interface.

2.2. Clouds Kernel Design

The kernel design has been through several design phases and is nearly complete. The design effort has produced a cohesive set of implementation guides to the entire Clouds kernel.

The current designs are based on assumptions about efficiency and ease of implementation that seem to be intuitively clear at this time. We may have to reiterate some design decisions and modify some strategies after more hard data is available from the implementation experience.

2.2.1. Kernel Implementation The Clouds kernel consists of several major subsystems: the object manager, which is responsible for mapping objects into virtual memory and invoking object operations (including the initiation of remote object requests); the process manager, which controls the slave process pool available on each node in a Clouds system and also supplies primitives for synchronization and process dispatch; the storage manager, which provides permanent storage for object data and paging storage for the virtual memory system; the communications manager, which is responsible for controlling inter-machine communications (currently via the ethernet); and the action manager, which is responsible for managing action events. The subsystems have been in various stages of completion, but have recently been integrated so a fairly complete, running version of the Clouds kernel exists. The current status of each of the subsystems is described below. For more details, see the attached technical reports ([Pitt85] and [Spaf84]).

2.2.1.1. Object Management The object management subsystem is almost completely coded and substantial sections have been tested. In particular, a primitive remote procedure call (RPC) mechanism has been implemented and tested. The complete RPC mechanism implementation awaits the implementation of the action management subsystem. Object mapping is implemented and is being tested with the virtual memory support provided by the storage management subsystem. Page fault handling will be done in tandem by the object manager and the storage manager. After the original fault is caught by the system, the object manager determines where the fault occurred (in a client object, in system space, or in a per process space) and makes a storage management call passing that information. Storage management is then responsible for selecting a physical page (through a call provided by the virtual memory system) and filling that physical page from the proper block on secondary storage.

The object invocation routines are being refined and implemented. Object and storage management are required to interact heavily to process an object operation call. Object management must first determine that the operation call is valid. It then initiates a search (possibly a network wide search) for the object. The storage management subsystem is responsible for determining whether the object exists on its local node and for activating the disk segment for the object if the object is found. Object management resumes control to initiate the operation call. The object management interface also provides the hooks necessary for the eventual presence of the action management subsystem.

Because of the cooperation required between object, storage, and action management, several iterations of the interface design were made before finally settling with the current design. It is felt that the current design meets all the requirements of the various subsystems involved.

The object manager and action manager normally supply certain special object operations, such as "create instance", "commit action", and "destroy action." Clouds programmers are able to reprogram these operations, so that in addition to performing necessary functions, the operations to the particular initializations, customized recovery operations, and cleanup that are specified. The object and action manager provide this support as part of the kernel interface, which can be accessed through the runtime system used by the Aeolus programming language.

2.2.1.2. Process Management The process manager is completely coded and tested. It provides a very rich set of synchronization primitives, which include semaphores, read/write locks, and general event mechanisms. Facilities for blocking with a time-out value are included. Code for the initialization of the slave process pool is running, as is that for dispatching processes. Slave processes are created at system initialization and are available for use as requests arrive. This accelerates the creation of processes for requests such as RPC's. The process management subsystem supports a primitive

round-robin scheduler with five priority levels.

2.2.1.3. Communication Management The communications management subsystem currently consists of the ethernet driver and associated software. This code has been tested and integrated into the Clouds kernel. The driver supports communications protocols not only for Clouds machines but also for machines running Unix 4.2bsd. Support for communication with Unix systems was implemented because it provides several possibilities supporting further development of the Clouds system. One such possibility is the development of a virtual disk for the Clouds kernel. Clouds kernel device requests to the storage manager could actually be handled by a disk running under a Unix system using the Clouds-Unix protocol on the Ethernet. This would provide either additional devices (very quickly, since the same interface at the Clouds end could be re-used) and also a facility for dumping status information for offline debugging. The communications subsystem recently was interfaced with the object management subsystem to provide a primitive working RPC mechanism.

2.2.1.4. Storage Management The storage management subsystem of the kernel consists of three classes of objects: devices, partitions, and segments. These object classes contain the structures and algorithms required to provide recoverable object data under the Clouds kernel. The subsystem is primarily concerned with the storage of on secondary storage devices, but is also necessarily involved in the management of virtual memory and action/object management.

There is a working device object (for the RL02 removable storage disk), that supports not only the conventional device operations (read and writes), but also provides a mechanism whereby the storage manager can insure that writes to devices performed by actions are done before the action completes. System failures will not catch object data in an inconsistent state. By having this support at such a low level, the storage manager relieves action management of some of its burden. In fact, the storage manager provides action management with a few simple calls that perform all the functions required to provide recoverable transfers of data to secondary storage.

The storage available on the RL02 device is not extensive (10 megabytes per pack) and the RL02 is not meant to be the primary drive for the Clouds system. However, it does provide a suitable testing device. A driver object is under development for another DEC drive, the RA81. This is a more sophisticated and larger drive (456 megabytes, fixed medium) than the RL02 and consequently the implementation of the RA81 object has been more complex than that for the RL01. The device object for this device is partially implemented and is being tested incrementally.

It should be noted that the design and development of the various devices discussed (and indeed those that will come later) have all been done using a standard interface to the Clouds system. This will allow us to bring new classes of devices onto the Clouds system with little difficulty. The only difference between the RL02 device driver

and the one being developed for the RA81 is that the RL02 object was written assuming only one such device existed (for simplicity). The RA81 object is being written for multiple drives per controller which will require some structural changes for the kernel, which have already been designed.

The device objects are being implemented with the idea that devices may be dynamically added to, removed from, and initialized on a running Clouds system. Currently, support is provided to allow operators to manually mount, unmount, or initialize devices, but further work could be done to automate this process.

The partition object is almost complete and running. Partitions can be created and removed from a device (dynamically, although support for this is not as clean as would be liked), read and write operations on partition blocks are available, storage may be allocated and deallocated from the partition, and the partition directory may add, remove, or locate items on the partition. The major component missing from the partition object is the partition activate call. This call brings certain partition structures into virtual memory and performs consistency checks on the partition data. Also the partition activation call initiates the action management cleanup that is performed by the storage manager. This processing is currently being integrated into the activation call.

The code for segment object is undergoing testing and final implementation. The segment object provides an interface to the storage management subsystem for the rest of the kernel. Primarily, the interface is through the abstraction of the segment type. The segment type is generally (though not always) simply a convenient alternate view of some client or system object (as a sequence of uninterpreted bytes). This allows the kernel to handle the various types of objects in a uniform manner. Coding necessary for handling segment level reads and writes and for handling page faults (in cooperation with object management) is complete and is in testing.

The segment object also provides the recoverability of object data. It provides the action management subsystem with a set of routines which transfer the data from virtual memory to secondary storage in a consistent manner. When invoked by the action management subsystem, the routines determine which parts of an object were modified by an action, and how to transfer the modified portions to secondary storage. The algorithms that do this are detailed in [Pitts85], along with an overall description of the storage manager. The algorithms described in the referenced report use the technique of shadowing current versions of a segment, making the shadow versions permanent on action commit. The data recovery routines are still under-going implementation at present. Since action management will not be available before the completion of the storage manager, testing of the recovery features of storage management will be done by simulating requests by the action management system.

2.2.1.5. Action Management The design of action management is complete. It includes several areas, but primarily it is concerned with the control of action events in the Clouds system. Other features included in the action management design are a simple algorithm for global object searches, the use of a global, kernel database, a time driven orphan detection mechanism (developed by Martin McKendry and Maurice Herlihy at Carnegie-Mellon University), a design for a global Lamport clock for the Clouds system which supports the orphan detection mechanism, and a design for a generalized locking facility for programming objects. Although coding has not started, the design includes enough implementation details so that this effort can proceed quickly.

The orphan detection algorithm mentioned above is quite different than that originally described in [Allc83a]. The new orphan detection mechanism attaches two time values to all action requests, in addition to the usual time-out value for deadlock recovery. These values are a quiesce time and a release time. After the quiesce time for an action has passed that action can initiate no further requests. The release time indicates the earliest time an action can release any locks that it hold. The release time is always greater than the quiesce time. Orphan are prevented from producing erroneous results by the eventual passing of their quiesce times. Generally, the period until an action's quiesce time is not long, requiring a refresh phase which increases first the release time and then the quiesce time of the action. This allows the action to continue work.

Action management's lock facility allows the creation of not only simple read/write locks, but also locks with more complex compatibility modes. For example, it is possible to create a lock with more than two modes and then specify how the modes conflict. In fact, one could create read/write locks in this fashion, but it is expected that the read/write locking style will be popular enough to justify a separate implementation. Also, locks need not be create for a specific instance of a structure, but may be defined over a whole domain of such structures. The data (a file, for example) need not exist at the time the lock is taken. This flexible locking mechanism, along with the redefinition of special object operations, allows the Clouds programmer to customize the recovery of the objects and action that are developed in addition to having available the default system recovery.

The search algorithm, as mentioned, is simple but attempts to do as much as possible to limit searches for objects. This is because in order maintain the location independence of objects from the sites on which they reside, the local object cannot determine the where an object is by examining the capability for the object. The object may be local or remote; invocations should be handled transparently. As the number of nodes in a Clouds system increases, the effort and time spent searching for an object could become quite significant. Therefore, information is kept in a global state database which aids in the search. The information in this low-level database is not guaranteed to be exact or complete. It does provide some hint of where the

object might be by maintaining several sorts of information; the last place an object was found, where the object was moved by the system, or even where the device on which the object last resided was moved. Always, as a heuristic, the node whose name is contained in the birth-place of the object is a high priority.

The global state database was mentioned as a source for clues for the object search, but it is actually more than that. Many types of system information is placed in the database, then to be propagated through the network using the algorithms described [Allc83a]. In particular, action state information, workloads, uplists, and other system information can be propagated in this manner.

2.2.2. Compiler Development

The systems programming language for Clouds implementation is Aeolus. Currently all development is being done on a VAX running Unix 4.2bsd, using "C". This is pending the full implementation and testing of the Aeolus Compiler. Once the compiler is implemented, and it interfaces to the Clouds system (the compiler generates objects), further development will use Aeolus.

The compiler implementation for Aeolus is currently underway. The Amsterdam Compiler Kit (ACK) is being used to generate code for both the Clouds system running on VAXen, and the Bubbles system running on 8088/8086 based systems (IBM-PC/XT/AT). Work on semantic routines for Aeolus is proceeding in parallel with the development of intermediate code for ACK. This work is being done in Pastel, an extended Pascal dialect.

2.3. Fault Tolerance and Probes

Use of probes in monitoring and fault tolerance is being studied. Probes are somewhat like messages, but unlike messages they are handled by traps handlers in processes and special probe handlers in objects. Thus probes can be sent to both passive as well as active entities. This gives rise to a powerful paradigm that is useful for a lot of activities, from monitoring, status enquiries to emergency messages. An application of probes to fault tolerant scheduling has been discussed in [McKe84c].

2.4. User Interfaces

The Clouds user interfaces are at several levels. The Clouds system runs a shell that allows hierarchical name spaces and common shell functions as the service routine for each user. The interface to this shell is via the intelligent terminals. This part of Clouds is still under the design phases.

The Human Factors group at Georgia Tech is looking at advanced user interfaces which will use the properties of "transitionality" to handle novice and advanced users at their own levels of sophistication. The transitional user interfaces will be built both at the

intelligent terminal level as well as the Clouds shell level.

2.5. Publications

The publications that have resulted from this research have been referenced below.

3. References

[Allc82]

Allchin, J. E., and M. S. McKendry, Object-Based Synchronization and Recovery, Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, September 1982

[Allc83a]

Allchin, J. E., An Architecture for Reliable Decentralized Systems, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also released as technical report GIT-ICS-83/23)

[Allc83b]

Allchin, J. E., and M. S. McKendry, Synchronization and Recovery of Actions, Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983

[Allc85]

Allchin J. E., Dasgupta P., LeBlanc R. J., McKendry M. S., Spafford E., The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System. (draft)

[Kenl85]8

Kenley, G. An Action Management System for a Distributed Operating System, Masters Thesis, School of ICS, Georgia Tech.

[LeBl85]

LeBlanc, R. J., and C. T. Wilkes, Systems Programming with Objects and Actions, Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, Colorado, May 1985 (also available as Technical Report GIT-ICS-85/03)

[McKe82]

McKendry M. S. and Allchin J. E. Object-Based Synchronization and Recovery Technical Report GIT-ICS-82/15, Georgia Inst. of Tech.

[McKe83]

McKendry, M. S., J. E. Allchin, and W. C. Thibault, Architecture for a Global Operating System, IEEE Infocom, April 1983

- [McKe84a] McKendry M. S. Clouds: A Fault-Tolerant Distributed Operating System. Technical Report, Georgia Inst. of Tech.
- [McKe84b] McKendry, M. S., Ordering Actions for Visibility, Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Spring, Maryland, October 1984 (also available as Technical Report GIT-ICS-84/05)
- [McKe84c] McKendry M.S., Fault Tolerant Scheduling Mechanisms, School of ICS, Technical Report, Georgia Tech.
- [Pitt85] Pitts, D.V. and E. Spafford, Notes on a Storage Manager for the Clouds Kernel, School of ICS, Technical Report, Georgia Tech.
- [Spaf84a] Spafford E. Kernel Structures for a Reliable Multicomputer Operating System Thesis Proposal, Georgia Institute of Tech.
- [Spaf84b] Spafford, E. and M.S. McKendry, Kernel Structures for Clouds, School of ICS, Technical Report, Georgia Tech.

**Notes on a Storage Manager
for the
Clouds Kernel***

Technical Report GIT-ICS-85/02

January 1985

Last Revision: October 20, 1985

*David V. Pitts
Eugene H. Spafford*

The Clouds Project, School of Information and Computer Science
Georgia Institute of Technology, Atlanta, Georgia 30332

* This research is funded in part by NASA grant NAG-1-430 and by NSF grant DCR-8316590

CONTENTS

1. Background	2
2. Hardware and Environment	4
3. The Design Approach	6
4. Device Objects	8
4.1 Device Media	8
4.2 Device Object Structures	8
4.3 Device Object Calls	10
5. The Partition Object	12
5.1 Partition Data Structures	13
5.2 Calls on the Partition Object	16
6. The Segment Object	18
6.1 Segment Object Data Structures	18
6.2 Calls on the Segment Object	18
7. Reliable Storage Management	21
7.1 Segment level recovery	22
7.2 Partition level recovery	26
7.3 Device support for recovery	28
7.4 Summary	29
8. Conclusions	30
REFERENCES	31

LIST OF FIGURES

Figure 1. Architecture of the Clouds kernel	2
Figure 2. Clouds hardware configuration	4
Figure 3. The system device table and other device object structures	9
Figure 4. The system partition table and other partition object structures	13
Figure 5. Two implementations of a Bloom filter	15
Figure 6. Clouds kernel segment structure	19
Figure 7. Actions block on competing commits	21
Figure 8. Precommitted segment	23
Figure 9. A committed segment	24
Figure 10. An aborted segment	25

**Notes on a Storage Manager
for the
Clouds Kernel***

Technical Report GIT-ICS-85/02

January 1985

Last Revision: October 20, 1985

*David V. Pitts
Eugene H. Spafford*

The Clouds Project, School of Information and Computer Science
Georgia Institute of Technology, Atlanta, Georgia 30332

Abstract: The Clouds project is research directed towards producing a reliable distributed computing system. The initial goal of the project is to produce a kernel which provides a reliable environment with which a distributed operating system can be built. The Clouds kernel consists of a set of replicated sub-kernels, each of which runs on a machine in the Clouds system. Each sub-kernel is responsible for the management of resources on its machine; the sub-kernel components communicate to provide the cooperation necessary to meld the various machines into one kernel.

This report documents a portion of that research, namely, the implementation of a kernel-level storage manager that supports reliability. The storage manager is a part of each sub-kernel and maintains the secondary storage residing at each machine in our distributed system. In addition to providing the usual data transfer services, the storage manager ensures that data being stored survives machine and system crashes, and that the secondary storage of a failed machine is recovered (made consistent) automatically when the machine is restarted. Since the storage manager is a part of the Clouds kernel, efficiency of operation is also a concern. We wish to reduce the overhead required to ensure the recoverability of secondary storage as much as possible, while adhering to the design goals associated with the storage manager.

* This research is funded in part by NASA grant NAG-1-430 and by NSF grant DCR-8316590

1. Background

In this section we present an overview of the Clouds kernel and discuss the philosophy behind its development. The Clouds approach to providing reliability is through the use of actions and objects, as discussed in [1], [2], [3], [4]. The Clouds kernel provides higher level applications such as operating systems with three primitives: *processes*, *actions*, and *objects*. An *object* is a typed collection of data which is manipulated by a set of operations. The data structures and the set of operations for the object define its type. An *action* is the unit of (fault tolerant) work in the Clouds system. Actions guarantee failure atomicity of the operations performed by them: the operation appears to either occur totally or not at all. *Processes* in Clouds are similar to processes found in other systems, and represent a thread of execution and control. Actions and objects are passive, waiting for a process to activate them. The model of computation for the Clouds system is that of a set of processes making operation calls on objects to perform services required by the system. In order to make these services reliable, the object operation calls are performed under the auspices of an action.

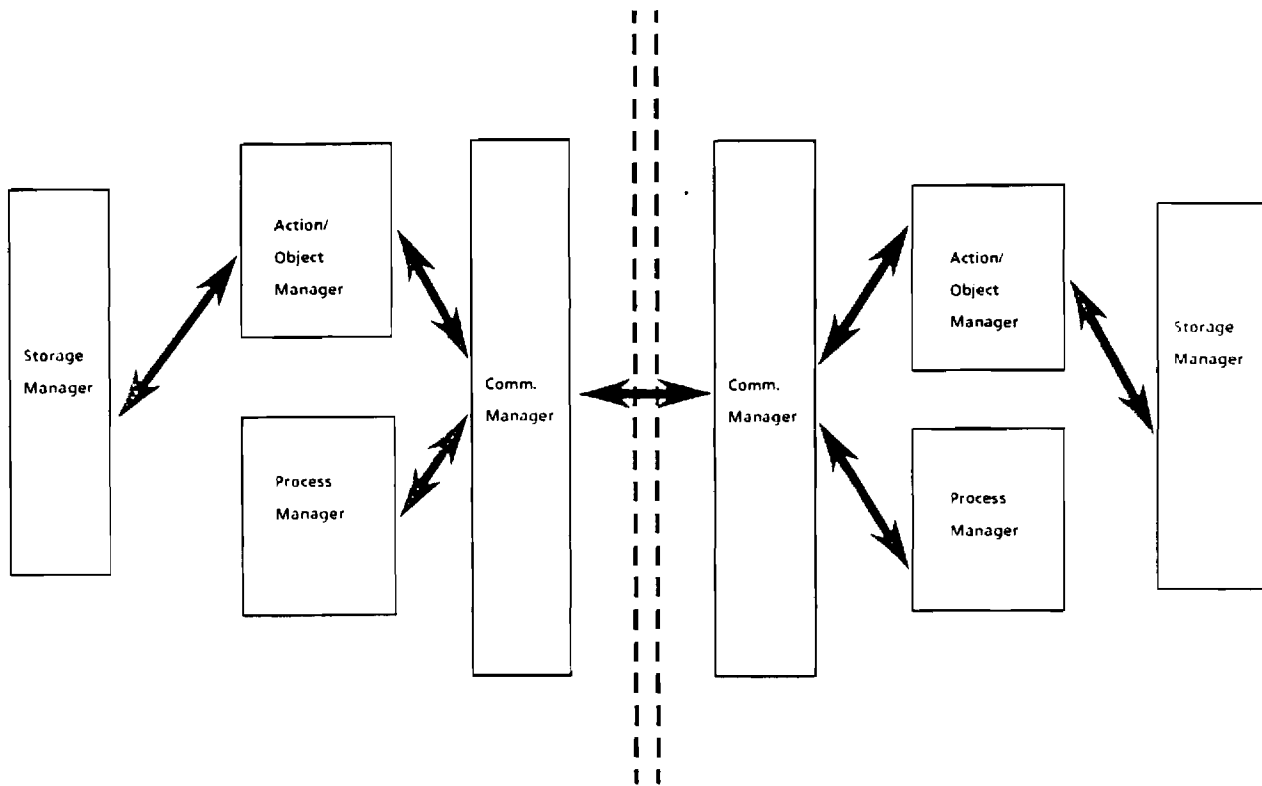


Figure 1. Architecture of the Clouds kernel

Clouds actions provide a mechanism that allows the programmer to violate the conventional notions of correctness and consistency, as defined by strict serializability, when programming reliable objects. The programmer can use any semantic knowledge about the intended activation of an object to program a customized method for providing the recovery of the object. This is done by the programmer writing new *abort* and *commit* operations for the object, which indicate how the object data must be recovered. By allowing object recovery to be customized in this way, we hope to provide increased concurrency in the execution of services compared to using the usual recovery and synchronization rules (i.e., serializability), and so improve the performance of the Clouds system.

The Clouds kernel has four major components: the object/action manager(s), the process manager, the communications manager, and the storage manager. Figure 1 depicts the architecture of the Clouds kernel for a system consisting of two nodes. The kernel is made up of two sub-kernels, one of which resides on each node that is part of the Clouds system. Each of the components of the kernel can communicate with its corresponding components on other nodes through the proper protocols.

The object manager is responsible for providing the object operation invocation mechanism. Each object is named by a unique capability comprised of a system name (called a *sysname*) and a series of rights which indicate which object operations are available to the invoking process. The object manager checks the capability provided by the operation call, locates the desired object instance, formats and maps the operation parameters, and activates the object. The object manager is involved with handling action bookkeeping, as necessary. The object manager also hides references to objects on other machines by providing a remote procedure call mechanism (RPC). The object manager makes an RPC look exactly like a local operation call.

The process manager creates, destroys, and dispatches processes. It manages local processes, as well as slave processes started to handle RPC's from other machines. The process manager is not a global scheduler; it simply manages local resources.

The communications manager is responsible for the transmission of information among the machines in the Clouds network. It maintains information about the connectivity of the network, the status of the various lines to which each machine is connected, and queues of outgoing and incoming data. The data that goes through the communications manager is uninterpreted — it might be an RPC or a part of a file that is being transmitted across the network. More detailed descriptions of the object, process, and communications managers can be found in ^[5] and ^[6].

The function of the storage manager was described above. It coordinates with the object manager to provide the correct commits and aborts of actions on object data residing on secondary storage. In the remainder of this report, storage will refer to the secondary storage (disk, tape, etc.) attached to a machine. Memory will refer to the main memory of the machine.

2. Hardware and Environment

The Clouds kernel is currently under implementation on three VAX 11/750's.¹ The machines have eight megabytes of main memory altogether and are interconnected via a 10 Mb/sec Ethernet. Also connected to the Ethernet are a set of IBM-PC's, which will serve as intelligent work stations. Future versions of the system may be connected by multiple networks of varying bandwidth.

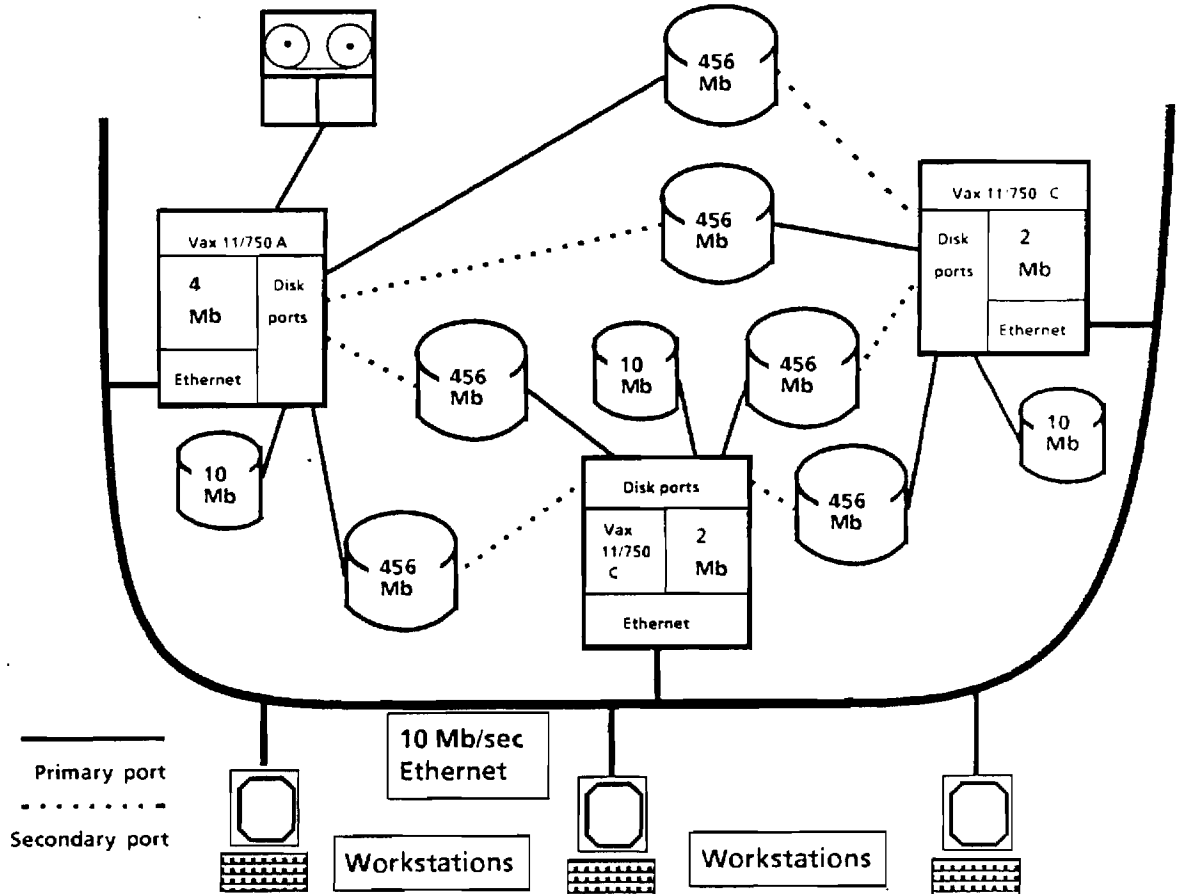


Figure 2. Clouds hardware configuration

Our prototype will have three types of storage devices available for the kernel. There may be a tape drive on one machine that will be used to archive data and perform conventional system backups. Each machine will have a RL02 removable pack disk drive, in which each pack provides 10 Mb of storage. We expect that RL02 media will be used as short term archive devices and boot devices. Finally, each system will also have up to four RA81 disk drives. Each such drive has a permanently mounted pack providing 456 megabytes of storage (unformatted). The RA81 drives are dual-ported; two controllers may be coupled to the drive simultaneously. However, the drive is on-line to only one of the controllers at any time. The switching of the device between controllers is done primarily by a front panel switch, but switching can be done under program control. The disks are controlled by UDA50 controllers

1. VAX is a trademark of the Digital Equipment Corporation

which use DEC's Mass Storage Control Protocol (MSCP). These devices are expected to provide the primary object storage for the Clouds system. Figure 2 shows the Clouds prototype system.

3. The Design Approach

The Clouds kernel provides user-defined objects² as the building blocks (along with atomic actions) of a reliable distributed system. The arguments for using an object-oriented approach in general, and as used in the Clouds project in particular, are presented elsewhere [7] and [3] and we will not repeat those rationales here. We feel that the kernel, in addition to supporting objects for higher levels of software, should also reflect the use of an object-oriented methodology in its design and implementation. To this end we have identified basic components of the kernel and isolated them as modules that are accessible only through a set of procedures defined for each module. These objects are then used to form the major systems of the kernel: the object manager, the process manager, the communications manager, and the storage manager.

We attempt to present kernel objects as typical Clouds objects that provide (restricted) access to functions and services provided by the kernel. However, there are differences between the objects that form the kernel and those that are supported by the kernel. The first such difference is in the implementation. User-defined objects will be created by users with an object-oriented language, such as Aeolus [8], [9]. This language enforces the use of an object-oriented methodology. Our kernel objects are currently implemented as C modules and most of the responsibility for adhering to the philosophy of object-oriented design is the responsibility of the programmer, not the programming language. Still, we believe the careful use of this object methodology despite the lack of support in the language provides benefits in the implementation and later maintenance of the kernel. It also may make the later conversion of the kernel to some other object-oriented language, such as Aeolus, more convenient.

The other difference reflects our concern for the efficiency of kernel functions and the operation invocation mechanism for objects. Many of the functions of the kernel are time-critical, or because of their frequent use require very efficient implementations. The operation invocation mechanism has overhead that we suspect cannot be afforded in these situations. Therefore, operation calls on kernel objects are handled differently than operation calls on user-defined objects. Calls on kernel operations may be performed as ordinary procedure calls or even as simple transfers to blocks of code. However, the appearance outside the kernel and the overall philosophy is that of an object-oriented kernel.

Some kernel objects are not generally available outside the kernel. For example, this is the case with the objects comprising the storage manager. Operating system code may occasionally require direct access to secondary storage, but it is hoped that for the most part the abstractions provided by objects will suffice.

The storage manager is based on three sets of objects: device objects, partition objects, and segment objects. Each of these objects manages the same actual item (secondary storage), but provides different abstractions. The device objects provide conventional device-level access to secondary storage. Partition objects allow devices to be sectioned logically according to the intended use of the storage on a device. Segment objects are the secondary storage extensions of the segment type provided by the kernel. Recoverable permanent objects are implemented at the level of segment objects.

The remainder of this report outlines a design for a storage manager for the Clouds kernel. It covers the important aspects of the structure and function of the storage manager, and discusses how the storage manager is used by and uses other parts of the kernel. The next three sections deal with the design and implementation of the device object, the partition object and the segment object. Those sections specify the data structures required plus the interface to the

2. Also referred to as *client objects*.

objects. Section 7 then covers how these objects are used by the kernel. In that section we discuss some of the issues related to the reliability of the storage manager and its relationship to the rest of the kernel. Section 8 contains a summary of this report, and a few conclusions and reflections on the storage manager.

4. Device Objects

As with conventional systems, the storage manager for the Clouds kernel provides a device level interface to secondary storage. This level of interaction with secondary storage is almost exclusively the province of the Clouds kernel. In fact, even within the kernel, most accesses to secondary storage are performed at some other (higher) level; only the storage manager makes frequent use of device objects.

4.1 Device Media

The storage manager views devices as two parts: the device itself and the medium currently being used by the device. This viewpoint is moot for fixed media disks, but for other forms of secondary storage, such as tape and removable disk storage, it provides additional flexibility in the configuration of a system. This separation is complete; a sysname exists not only for each device in use on a system, but also for each medium. However, in many cases the distinction between accessing specific media and accessing devices is not important, so we wish to hide this separation. Therefore, the storage manager provides a mechanism for binding a medium to a device.

Bindings between media and devices are generally performed at the initialization of the system and involve the association of device and medium. Binding a medium to a device may also involve the formatting of the medium. In this latter case, a new sysname is generated for the medium. This formatting or initialization of a medium will destroy any previous information that existed on the medium. The old sysname will no longer give access to any medium. The formatting of a blank or obsolete medium includes initializing the tables and structures that the storage manager requires. These structures are discussed in section 2.1.

In other cases, an existing medium is bound to a device. An existing medium is one which has a sysname and is formatted. The binding will involve the reading of the sysname from the medium and comparing it with the sysname passed to the storage manager. The binding will take place only if a match occurs. We are not attempting to address security issues with this design. Our interest is to provide flexibility, while maintaining some control over what is accessible. The use of sysnames to access media provides this control.

Once a medium has been bound to a device, any reference to the device refers to the bound medium. The usual sort of device calls then need only refer to the device. This device-medium binding stays in effect until it is explicitly broken by the storage manager.

In addition to setting up this correspondence between device and medium, this binding also initializes an instance of a storage object in memory. In particular, critical tables and other structures required by the device are brought into system memory. We will now look at the data requirements of device objects.

4.2 Device Object Structures

The storage on a medium is presented as a sequence of 512-byte blocks that are addressed by offsets from some fixed block. The offset used to address a block is called a device block number (DBN). As we shall see in section 5, this storage can be subdivided into partitions, allowing the storage on a device to be apportioned for policy reasons. At the device level, though, the storage manager deals only with a contiguous string of blocks; partition boundaries are not visible.

The device object is responsible for the transfer of data between secondary storage and memory. The device requires two tables in order to function. The first such structure is the media header. This table contains basic information about the medium and the device using it. This information includes the medium and device sysnames, the amount of available storage on the medium, and specifications for the device to which the medium is bound.

The other major structure is the index table. The index table describes the partitions that exist on this device. This will include information such as the location, extent, and type. The partition table is 16 entries long. Partitions are discussed in section 5.

The medium header and index table must be resistant to failures — in particular, device failures such as head crashes. If the index table is destroyed by a head crash, for instance, we lose access to the partitions on that medium. We therefore provide copies of the tables, placing the copies on different cylinders in order to minimize the risk from multi-sector failures. The alternate copies will be located in known positions based on some computable function. We do not anticipate problems as far as maintaining the consistency of the slave and master versions of the table is concerned, since the tables are infrequently modified and any such modifications are generally done during the system initialization.

In addition, the device objects will maintain a structure in memory called a flush table. The use of this table is discussed in section 7, but briefly, the flush table allows a device to associate an action sysname with a set of requests. This supports the commit operation performed on recoverable objects. Some devices may require the device object to provide bad sector recovery. Objects written for these devices will have to maintain a bad sector table on disk.

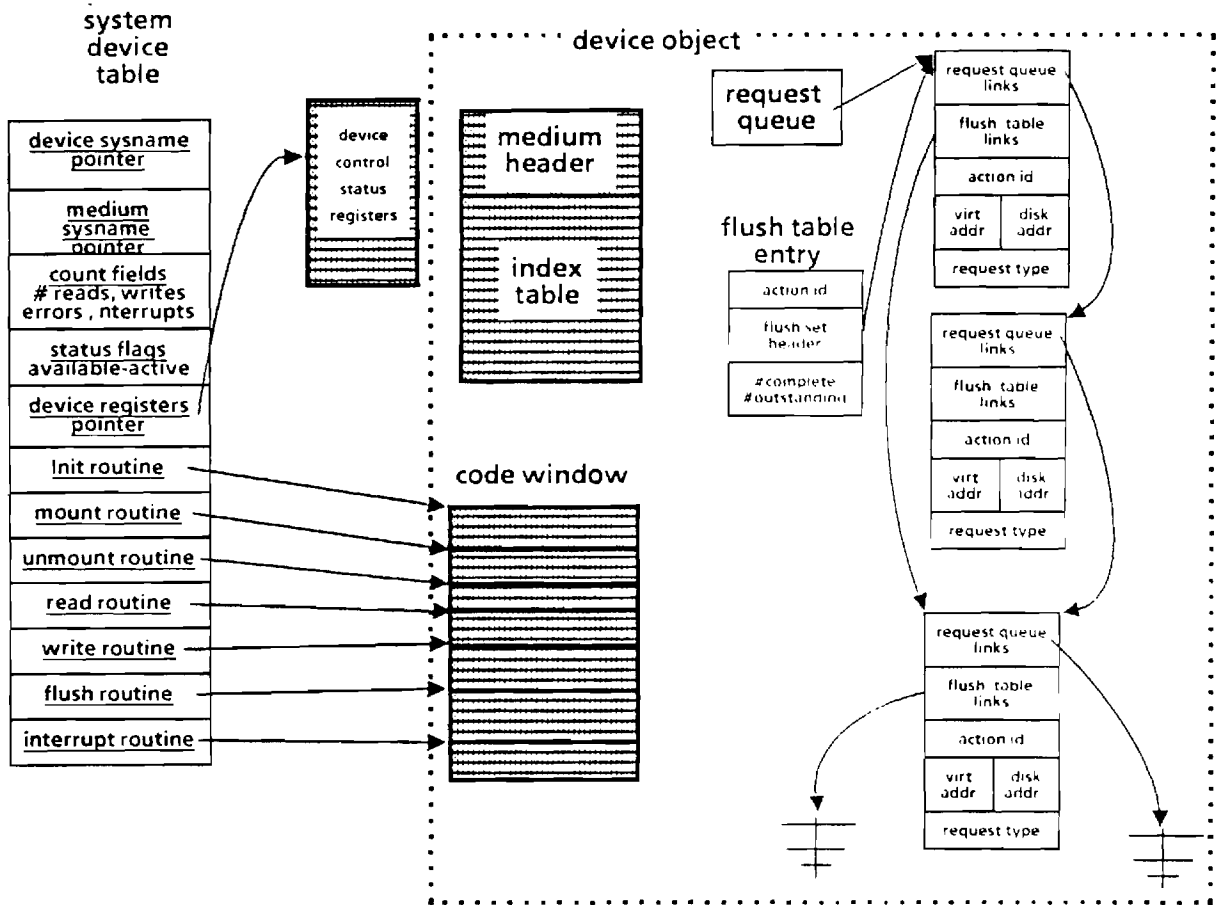


Figure 3. The system device table and other device object structures

The device object uses one other structure, the system device table. The system device table is not a part of the device object proper, but is actually the mechanism for managing the various instances of the device objects. This table lists all secondary storage devices that are active on the local machine. The device table entries contain pointers to device and medium sysnames, status variables for the device, device registers, and entry points into the operations for the

device object. Figure 3 shows a device object pointed to by one entry from the system device table.

4.3 Device Object Calls

The device object calls deal with the transfer of information to and from the device and with the relationship of the device to its medium. This allows for devices switching the physical medium used for storage in a uniform way. Device and media sysnames are generally needed by those calls setting up a binding between the medium and the device. Calls which perform i/o do not require a sysname. The proper device object calls are found through the system device tables.

4.3.1 *init(devname) return medname*

Init generates a sysname for the medium currently on the device specified by **devname**. This sysname is written in the medium header. Also written into the medium header is the device specific information that is required. An area for the medium index table is reserved. The return value is the medium. This is a format call; any existing structure on the medium is overwritten. No other formatting is done, however. Any partitions desired are created later by other calls. Redundant copies of the medium header and index table are created for reliability.

After the medium has been formatted, **init** mounts the device. See the description of **mount** for details.

4.3.2 *mount(devname, medname) returns integer*

This call binds the device called **devname** to the medium called **medname**. The sysname presented to the call is compared to that in the medium header. If the two match, the device and medium are bound. The medium index table and the medium map table are read from the disk. If the device requires it, a bad sector table is created from the device. The return value specifies the status of the call (success, failure).

4.3.3 *return_medium_cap(devname) returns medname*

This call returns the sysname of the medium that is bound to the device named **devname**. The return value is this sysname. If the device is unbound, a valid sysname might still be returned if a formatted medium is present in the device. In this case, the call can be seen as an operation to read a label.³ This allows us to use media for which all currently existing copies of the sysname are deleted or unavailable.

4.3.4 *unmount(devname) returns integer*

Unmount breaks a device/medium binding. All partitions on the medium are de-activated. The return value is the status of the call.

4.3.5 *read(lbn, address) returns integer*

This call transfers the contents of a record located at disk address **lbn** to the page in memory at **address**. **Read** blocks the calling process on a semaphore until the request is complete and returns an integer indicating success or failure of the request.

4.3.6 *write(id, lbn, address, flag) returns integer*

This call transfers the contents of a page in memory at location **address** to the record located at address **lbn** on the device in question. **Id** is an identification used to associate this request with a set of requests being issued by an action. If **id** is an action sysname, then the device object looks the action id up in a flush table and if it is not there, creates an entry for the action and the request; if the action id is in the table, the request is added to that entry. If **id** is zero, then there is no action associated with this request. **Flag** is used to indicate whether this is a forced

3. This kind of operation might seem to present a security hole in the system, in that it allows the system to determine the name of an unknown medium and then mount it. However, note that this call can only be executed by kernel code or by a user call with special kernel capabilities, and these are assumed to be trustworthy.

write. If flag is non-zero, the device interrupts the normal scheduling of requests by placing this request at the head of the queue. The new request is performed immediately after the current request is completed. A forced write blocks the calling process on a semaphore until the request is complete. Non-forced writes do not normally block the caller.

4.3.7 flush(id) returns integer

Flush uses the flush table maintained by the device object to ensure that all write operations associated with the action identified by **id** are actually completed. The return value indicates the status of the call. A positive return value (the number of requests completed) indicates a successful flush. A zero or negative return value indicates that the action's sysname was not found in the flush table or that some error occurred while attempting to flush the specified requests.

4.3.8 enter(partname, size) returns lbn

Enter registers a partition on the device. This involves making an entry for the partition in the index table for the device, placing the partition sysname, **partname**, and the partition size, **size**, in the entry, and allocating storage on the medium for the partition. The starting logical block number for the partition is placed in the index table and is returned as the value of the call. A negative return value indicates that an exceptional condition occurred, such as not enough storage for the partition on the device. **Enter** is called as part of creation of a partition.

4.3.9 remove(partname) returns integer

This operation allows the caller to remove a partition from the device. **Partname** is the sysname for the partition. The entry for the partition is removed from the index table on the device and the storage for the partition is deallocated. The return value indicates success or an exceptional condition, such as a non-existent partition. **Remove** is called as part of the removal of a partition from the device.

4.3.10 partitions(partarray) returns integer

Partitions places the partition entries in the device's index table into the array parameter **partarray**. The major use of **partitions** is expected to be at system initialization, where it passes partition's sysnames to the boot code so that the partitions may be activated. The return value is either the number of partitions written into the **partarray** (a non-negative value) or a negative value indicating an exceptional condition, such as a bad index table.

5. The Partition Object

The partition object represents an intermediate level of abstraction of secondary storage. Partitions are consecutive blocks of secondary storage that reside completely on one device. Each partition is a logical object in that it manages the allocation of its own storage and maintains the structures used to do so. A Clouds partition does not enforce any logical organization of the data which resides on the partition, at least not in the sense of a Unix⁴ partition. A Unix partition represents a separate file system and all the files on the partition have a hierarchical relationship. The objects residing in a Clouds partition may bear no relationship to each other. It is simply an administrative organization imposed by the partition system indicating how storage in a particular partition is managed.

The two important types of partitions are recoverable and non-recoverable. When a partition is made non-recoverable, it is a declaration that no recovery will be provided for object data stored on that partition and that recoverable objects should not be placed in it. There is no similar restriction for recoverable partitions; such partitions may contain a mix of recoverable and non-recoverable objects. Other partition types include those used for paging surfaces and those reserved for temporary items.

Partitions manage storage as device independent blocks of storage and these are the smallest units of allocation that partitions support. The blocks are addressed by a partition block number (pbn) which is an offset from the beginning of the partition. All partitions are a multiple of this block size.⁵

The partition has as its initial block a header containing partition specifications. The header repeats most of the information found in the medium index table entry for this partition, plus information about the partition's state. This structure is generally accessed only when the partition is activated or some change is made to the partition; at other times the information is in memory and is referenced there.

Another structure used by the partition object is the system partition table. Like the system device table, the SPT is not part of any one partition object instance, but is part of the underlying mechanism. The table contains entries for all partitions which reside on the local machine. Each entry in the table associates a partition sysname with the data structures and information for that partition. These structures and information include the starting block number for the partition, pointers to in-memory structures and buffers used by the partition object, and a pointer to the device object on which the partition resides. This last pointer is actually a pointer into the system device table. Figure 4 shows the complete relationship amongst these structures.

Another function of the partition object is to maintain the location of segments and make available this information upon request. One of the features supported by the Clouds kernel is the location independence of objects (and thus segments). We mean by this that an object may reside on any partition on any node in the Clouds system and may be moved to any other partition on any other node. This implies that each access to an object requires that a (potentially) system-wide search be initiated. The sysnames given to objects give no (definite) information as to the location of the objects. As can be imagined, such searches can be time-consuming. In particular, searches on the partitions at a node might require one or more disk access each. We discuss one method of lessening the impact of these searches shortly.

4. Unix is a trademark of AT&T Bell Laboratories

5. The preliminary implementation will undoubtedly make this size equal to the size of a main memory page frame.

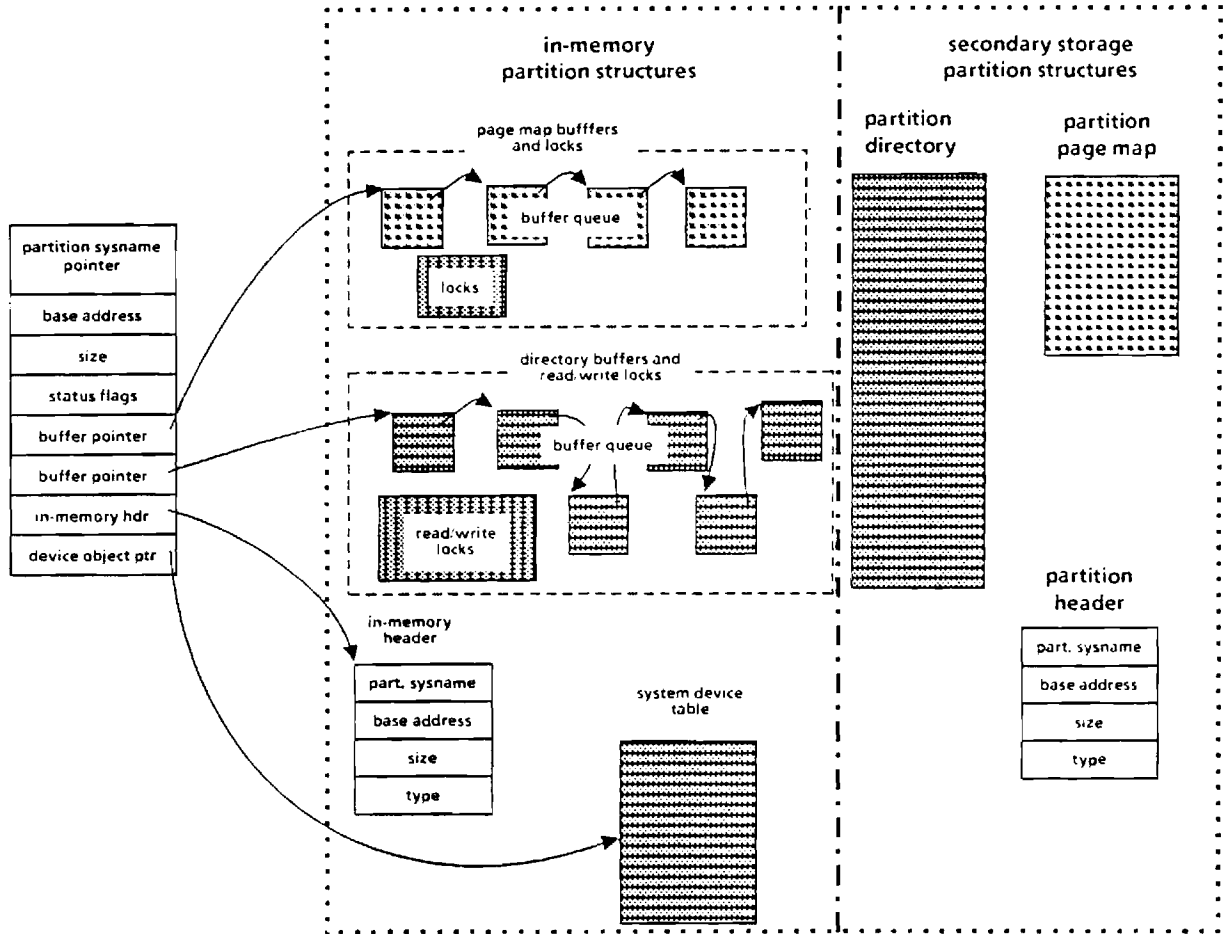


Figure 4. The system partition table and other partition object structures

5.1 Partition Data Structures

Two of the major functions provided by a partition object are the location of objects and the management of storage. To provide these functions the partition object maintains two structures: the partition directory and the partition page map. The partition directory is a large hash table which is composed of page-sized buckets. In our current implementation the bucket size is 512 bytes, allowing each bucket to hold sixteen entries, each entry consisting of sysname-pbn pairs. The sysname is the id for a segment and the pbn is the offset of the segment within the partition. The entries to the directory are hashed to the proper bucket on the sysname and then to the proper entry in the bucket by a secondary hash function, also on the sysname.

The page map is simply a bit map representing the storage allocation for the partition. This structure, along with the directory, contain most of the information that composes the partition state. As such, they are crucial to maintaining the reliability of the partition and the system as a whole, and some care must be taken in the modification and access of the partition directory and page map, as explained in section 7. Additionally, the storage manager must protect these structures from device failures. The basis for this protection is redundancy of the information. The partition directory and page map have duplicates at known locations in the partition. We are not overly concerned with the extra storage required; we calculate that even with duplicate structures we can keep the storage requirements for these two structures below one per cent of

the total storage. Combined with the protocols we follow for maintaining the reliability of segments and partitions, we should be able to minimize the access overhead caused by this redundancy.

The partition directory and page map may be too large to completely reside in memory and, in fact, we will not have them mapped entirely into virtual memory. Instead, we will maintain buffer areas for the two structures, bringing in new pages from secondary storage as needed, and using a least-recently-used discipline for replacement. We suspect that locality for the page map will be fairly good so that allocations of storage can be done from the memory buffers. However, we suspect that accesses to the partition directory will typically take one access to secondary storage. If our hashing functions are chosen properly we may be able to handle directory requests in (at most) one secondary storage access.

The partition object maintains another structure which it uses to avoid unnecessary secondary storage accesses altogether (or at least make such accesses rare). The structure in question is a Bloom filter ^[10] which we have called the *Maybe Table*. The Maybe Table is a probabilistic membership checker. It will indicate either that the object in question definitely does not reside on the partition being checked, or indicate that it possibly does. Thus, the Maybe Table gives a method of short-circuiting secondary storage accesses in cases where it gives a negative response. However, a positive response may still lead to unnecessary accesses to secondary storage. The key to success is to reduce the ratio of non-resident positive responses to all positive responses to as small a value as possible.⁶

As described in ^[10], a probabilistic membership checker is a hash table where collisions are allowed. There are two techniques described in that paper that present methods that could be used with Clouds object sysnames. In the first technique, the Maybe Table consists of a table of transformed entries. The transformation is a hashing function which takes a 48 bit sysname and produces a shorter Maybe Table entry. Several sysnames may hash to the same entry value. This entry value is then placed in the Maybe Table by the use of another hashing function; this time collisions are handled in a conventional manner. To query the Maybe Table, the sysname is once more transformed with the first hashing function, and the proper entry located using the second. If the retrieved entry matches the transformed sysname, a positive response is returned. Otherwise, the collision handling mechanism is invoked and another entry is tested. If a positive response has not been returned upon termination of this procedure, a negative response is returned.

A second scheme is to treat the Maybe Table as a bit-string and use t different hashing functions, each of which returns an index into the bit-string. Placing a new entry in the Maybe Table requires setting the bit whose index is returned by each hashing function. The test for membership requires that all bits whose indices are returned by the hashing functions be set; any clear bit causes the return of a negative response. Figure 5 illustrates the use of these two techniques. In the example, the Maybe Tables are 18 bits in length. In each case, sysnames are represented by three bits in the Maybe Tables. In the first case, sysnames are represented straight-forwardly by three bit entries; in the second case, three bits are set for every sysname belonging to the table.

The benefit drawn from the use of a Bloom filter such as the Maybe Table is that it is a more compact representation of the universe in which membership is being tested. In the case of the Clouds kernel, this is the sysname population of a partition. This allows more of the table to be kept in virtual memory (perhaps all of it), and so queries on the Maybe Table can generally be

6. This is an area that is open to further research. We believe that the goal is achievable by careful selection of the (possibly more than one) filters used, and their manner of implementation. We hope to do some measurements and research on this once the system is working.

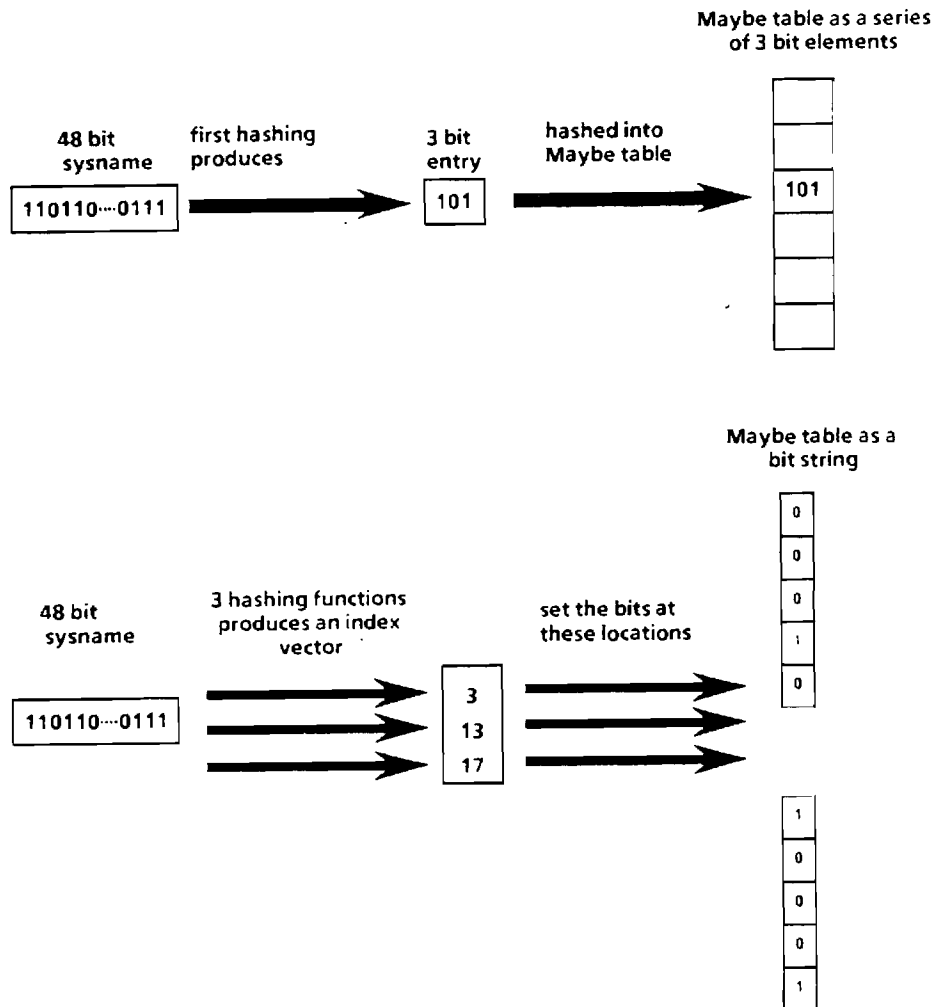


Figure 5. Two implementations of a Bloom filter

answered without going to secondary storage. If the response is negative, an unnecessary access to secondary storage is avoided, speeding the search for the proper segment. If the response from the query is positive, then an access to secondary storage is required, to either locate the segment or to ascertain that it is really not on this partition.

Maintaining the Maybe Table has several costs that must be considered. One, of course, is the initial creation cost. The storage manager will perform this initialization at system start-up for each partition and thus the time spent can be ignored. Another cost arises from the dynamic nature of the Clouds system. Objects are created on a partition, deleted from the partition, and moved to other partitions. Clearly, these changes must be reflected in the Maybe Table else the performance will be degraded. Creation of objects and the movement of objects onto a partition pose no problem: the sysname can simply be incorporated into the table via the methods described above. However, deletions of objects and movement of objects from a partition are more troublesome. An entry or set of bits in the Maybe Table cannot be cleared to remove a sysname's presence from the Maybe Table because several sysnames may be represented by the same entry or set of bits.

The simplest solution is to simply reconstruct the Maybe Table at intervals during the lifetime of the system. This reconstruction may be done asynchronously as a background task. The question of when the Maybe Table should be rebuilt is not yet answered. It would seem best to base the interval between reconstructions on activity of the partition, particularly the rate of deletions. This could be done indirectly by recording the performance of the Maybe Table and reconstructing the table when the performance falls below a given threshold. Or the monitoring could be more direct, measuring the number of deletions and movements of objects from the partition. Both of these methods have advantages and disadvantages. The indirect method for example, seems to be desirable since it measures the attribute that we want to optimize (avoiding disk accesses). However, a burst of queries for a sysname not resident on this partition but which happens to hash to the same entry or set of bits could cause a severe drop in performance even though the table as a whole is behaving reasonably well.

We are currently incorporating a Maybe Table into the partition object as described in ^[10]. We wish to get the maximum performance from the Maybe Table with the minimum impact on virtual memory. Therefore, we may consider other implementations for the Maybe Table, depending on the performance obtained. It may be, for example, that we are able to take advantage of the nature of the sysname population to improve the performance of the table.

5.2 Calls on the Partition Object

The storage management system uses the following calls to manipulate the partition data. Most of the calls require at least one sysname as an input parameter, usually a sysname for the partition (the exception being `create_partition`; see below). Occasionally, sysnames for segments and devices may also be required.

5.2.1 *P_create(devname, size, partatt) returns partname*

P_create reserves a sequence of records on a device to form the partition. **Size** is the size of the partition in bytes (this parameter is rounded by the call to the record size of the device) and **devname** is the sysname of the device on which the partition is to reside. A sysname for the new partition is generated and returned as the value of the call. The record location of the initial record of the new partition is stored, along with the size (in device records) and the partition sysname, in the media index table. The attributes of the partition, specified in the input parameter **partatt** are also stored in this new partition entry. **P_create** makes use of the **enter** call on the device object to perform its task. In particular, **P_create** must be able to request allocation of storage from the device.

5.2.2 *P_destroy(devname, partname) returns integer*

This call takes the two sysnames given as input parameters and frees the chunk of storage used by the named partition. **partname** specifies the particular partition to be destroyed and **devname** specifies the device on which it resides. The integer return value indicates the status of the partition after the call (destroyed or not found on this device). The call removes the partition's entry in the **media index table** and releases the storage used by the partition. The device manipulations are performed with the device object call **remove**. **P_destroy** also makes calls on the device object to perform its task.

5.2.3 *P_enter(partname, segname, pbn) returns integer*

P_enter places an entry in the partition directory for a segment. **Segname** and **partname** identify the segment and partition, respectively. The entry in the directory includes the segment sysname and the partition block number, **pbn**. The call also modifies the Maybe Table. The return value indicates success or an exceptional condition.

5.2.4 *P_remove(partname, segname) returns integer*

This call removes the entry for a segment from the partition directory. **Segname** and **partname** identify the segment and partition, respectively.

5.2.5 *P_return(partname, segname, seginfo) returns integer*

P_return returns the segment header indicated by **segname** which resides on the partition specified by the input parameter **partname**. The header includes the sysname for the segment, the size of the segment (in partition records), the record address of the segment header, and whether the segment is recoverable. The segment header is placed in the parameter **seginfo**, which is a pointer to a block of storage reserved for the information. If the segment is present, the return value of the call is positive; otherwise the return value is negative. The call finds the information by searching the partition sysname map and examining the segment header found. The Maybe Table is first queried in an attempt to avoid unnecessary secondary storage accesses.

5.2.6 *P_get_{first,next}(partname, number, segarray) returns integer*

These two calls are similar to **P_return**, in that they return the attributes of a segment found on the partition specified by the input parameter **partname**. The segment is unspecified, however. **P_get_first** places the first **number** of segment sysnames appearing in the partition directory in the parameter **segarray**. **P_get_next** can then be used to retrieve the attributes of the **number** subsequent segments. The two calls share a static variable which holds the index of the next segment about which information will be returned by **P_get_next**. The variable is reset to zero after the last entry in the partition directory is accessed and is initially set to zero. which is an array large enough to hold the requested number of sysnames. The return value is either zero, indicating no sysnames could be found, or the number of sysnames actually returned by the call.

5.2.7 *P_available_space(partname) returns integer*

This call simply returns the number of free records on the partition indicated by **partname**. A negative value may be returned in exception conditions. The call does a bit count on the volatile record map. Because the volatile free map contains allocations and deallocations for uncommitted actions and because no synchronization is done on the record map, the value returned should be considered only an approximation of the "true" number of free records.

5.2.8 *P_{read,write}(partname, part_offset, address) returns integer*

P_read causes the transfer of the contents of a partition record, **part_offset** from the partition specified by **partname** to the physical page in memory indicated by **address**. **P_write** reverses the procedure, transferring the contents from the physical memory page to the partition record. The calls use their return values to signal exceptional conditions. The virtual memory system uses this call to handle page faults.

5.2.9 *P_getblk(partname) returns pbn*

P_getblk simply returns the partition block number of a free page on the partition. The volatile page map is updated to reflect the allocation. A negative value is returned if there is no partition storage remaining.

5.2.10 *P_returnblk(partname) returns integer*

This call deallocates the page at the partition block number passed through **pbn**. The volatile page map is updated. A negative value indicates a bad partition block number.

5.2.11 *P_restore(partname, pbn) returns integer*

The **P_restore** operation is called on system startup to examine the partition. If necessary, the operation will perform any repairs to the partition structures required to bring it back into a consistent state. The call will also cleanup any unfinished action processing. This sort of repair is done on a partition-by-partition basis, since not all partitions have the same attributes and therefore will not require the same processing. In particular, cleanup of action processing is not necessary on partitions not supporting recovery and partitions being used as paging surfaces. **P_restore** must determine attributes of the partition by examining the partition header and then proceed accordingly. The details of **P_restore**'s operation are described in section 7, which is concerned with the reliability of the storage manager. **P_restore** also initializes structures used by the partition object, such as the Maybe Table.

6. The Segment Object

The segment object provides the final level of abstraction for secondary storage. With these objects, we are operating on blocks of storage allocated by the partitions. The abstraction provided by the segment object is that of a sequence of bytes (kernel segment type). The implementation is actually a tree of fixed length blocks of storage, as we shall see.

Segment objects provide a standard abstraction for the kernel to manipulate and process all Clouds objects. The object implementation provides mechanisms for mapping segment data in and out of virtual memory, creating and destroying segments, and modifying segments. The necessary algorithms for maintaining the reliability of the segment data exist at this level.

The segment object is unconcerned with the internal organization of the objects it is managing. The storage management system treats segments as uninterpreted bytes. Any interpretation is performed by other parts of the kernel, such as the object manager.

6.1 Segment Object Data Structures

Recall that a partition directory has a set of entries which contains the pbn for the segments residing on the partition. The partition block addressed by one of these entries contains a segment header that identifies the segment. The complete header is 512 bytes long and contains the segment (object) sysname, the object type sysname, a segment status field, a segment shadow pointer (the status field and pointer are used for recovery), and the size of the segment in bytes. The remainder of the header contains an array of pointers which lead to the segment data. These pointers address one of two sorts of blocks: index blocks, which are arrays of pointers to other blocks, and data blocks, which actually contain segment data. If, however, the storage required for segment data is less than that used for the array of pointers in the segment header, the segment data can be placed in the segment header itself. This would provide for the efficient processing of very small segments. Figure 6 shows the segment structure.

A segment is a tree whose depth depends on the amount of data in the segment. Hence, the smallest segment may have a depth of two (the header and the data blocks addressed by the header), but trees of arbitrary depth are supported. This also means that occasionally the segment will be restructured when its size is increased.

The interaction of the segment system and virtual memory is still being designed. It should be pointed out that much of the manipulations performed by the segment object will involve the segment's representation in virtual memory and the structures maintained by virtual memory itself. The segment system also makes some assumptions. One of these is that the location of the segment is known. That is, the action or process using the segment knows the partition on which the segment resides. Particularly, most segment calls do not require a partition sysname as a parameter.

6.2 Calls on the Segment Object

The following calls all require the sysname for the segment being manipulated. Any offsets are data record offsets, using the logical view of the segment.

6.2.1 *S_create(partname, segname, attr)* returns integer

S_create allocates storage for a segment and sets up the segment header and index records. The input parameters are the two sysnames for the partition and segment to be created,⁷ and a structure holding information about the segment (its size, object type, recoverability). The storage for the segment can be allocated and structured on the basis of the size field of *attr*.

7. Note that this call does not return a new sysname for the segment. If that were the case, it would not be possible to move existing segments into a partition and still reference them by their old names.

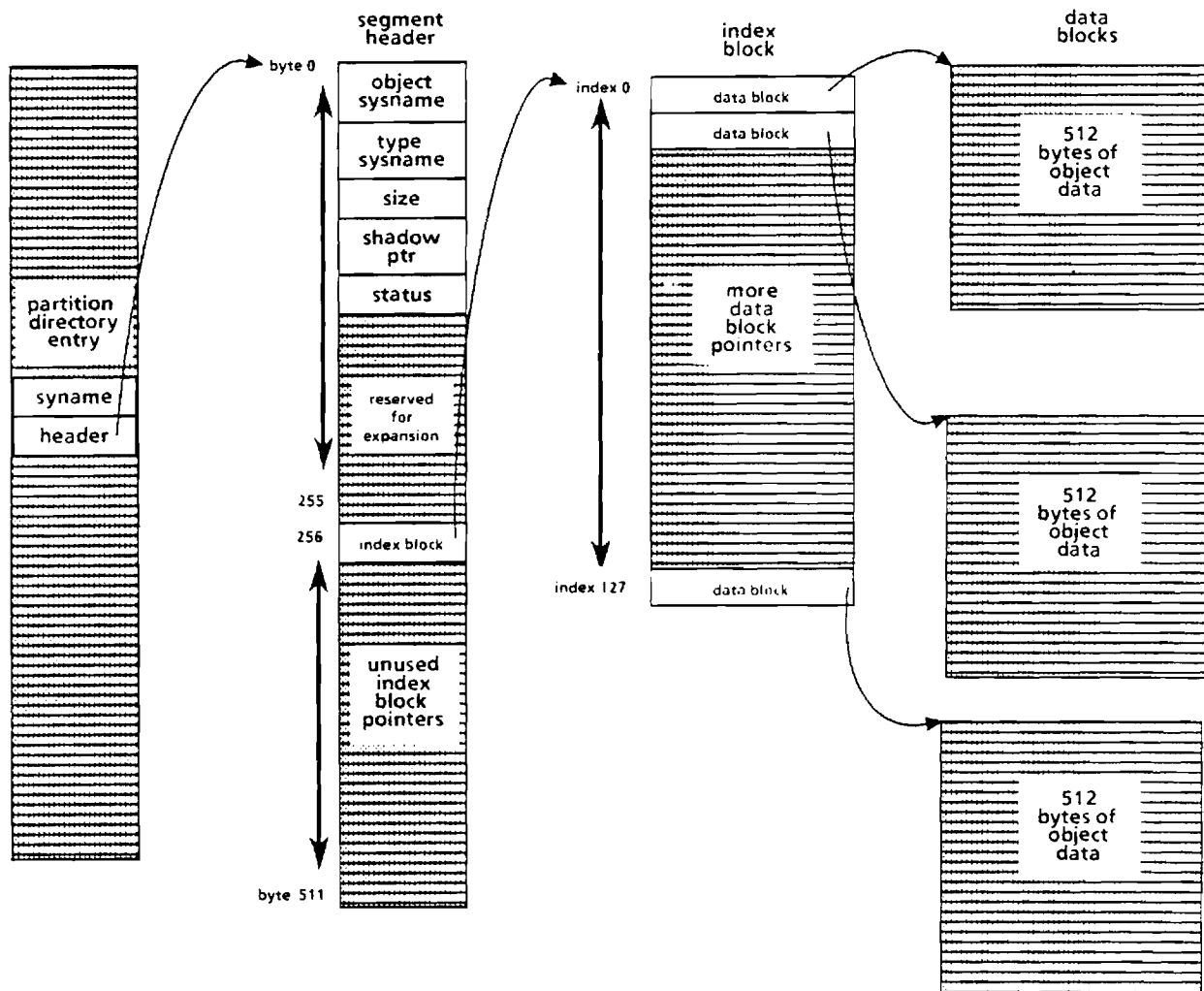


Figure 6. Clouds kernel segment structure

Data records are written in subsequent requests. The return value indicates the call status.

6.2.2 *S_destroy(partname, segname) returns integer*

This call deallocates storage for a segment. The sysname for the segment, **segname**, is removed from the partition directory.

6.2.3 *S_read(segname, offset, size, addr) returns integer*

The **S_read** call causes the transfer of **size** number of pages from storage to memory. **Segname** identifies both the memory and storage versions of the segment. The source of the pages is at location **offset** of the segment named by **segname**. **Addr** is the virtual memory address of the transfer destination. The return value indicates the status of the call.

6.2.4 *S_write(segname, offset, size, addr) returns integer*

S_write transfers data from memory to storage. **Addr** is the source of the transfer, in this case a virtual memory address. **Segname** is the sysname for the object (segment) whose data is to be transferred. Note that this identifies both the memory pages (source) and the secondary storage pages (destination) that must be transferred. **Size** number of pages, beginning at offset **offset** of the segment, are copied from virtual memory to the storage segment. The return value indicates the status of the call.

6.2.5 S_precommit(aid, touchlist) returns integer

S_precommit performs the segment level precommit protocol as described in section 5. **Touchlist** is a list of the objects which have been modified by the action. **Aid** is the sysname of the action making the precommit call. The call return value indicates the success or failure of the call.

6.2.6 S_eoa(segname, flag) returns integer

This operation performs the segment level commit or abort protocol as described in section 5, depending on the value of **flag**. The return value indicates the success or failure of the operation.

6.2.7 S_chgsize(segname, delta) returns integer

The call allocates or deallocates storage from the end of a segment. **Delta** is the number of records to allocate or deallocate (positive or negative value, respectively). The return value is the status of the call.

6.2.8 S_status(segname) returns integer

This call determines the state of a secondary storage segment by examining the status field of the segment header. The return value is this status (permanent, shadowed, precommitted).

7. Reliable Storage Management

In this section we look at the techniques used to ensure the reliability of the storage manager in the presence of machine failures and action aborts. All the techniques described below require the information and features provided by the use of atomic actions. This information includes the knowledge of when it is correct to make the effects of an operation permanent and what data has been modified. The storage manager provides a set of protocols that use this information to make the correct updates to secondary storage so as to leave the storage system in a consistent state. In order to understand these techniques and the motivation behind them, we need to understand how the Clouds kernel manages actions.

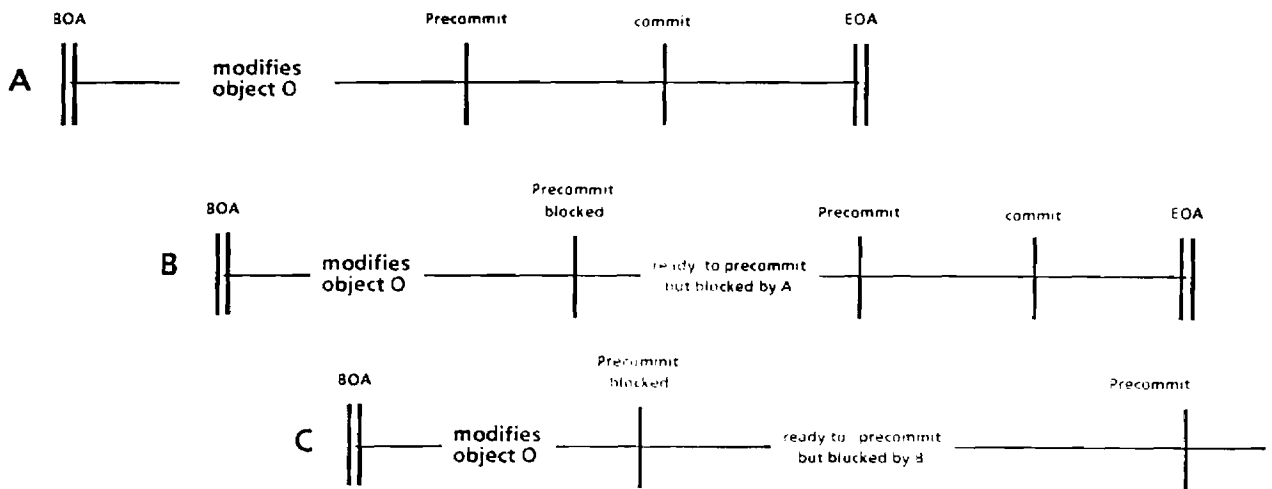


Figure 7. Actions block on competing commits

The Clouds system considers actions to be units of work. Many actions may be active in the same object, with each action updating object data. The only restriction enforced by the kernel⁸ on the synchronization of actions which are operating concurrently on a single object is at action precommit. An action that precommits in an object blocks all other actions from precommitting in that object until the precommitting action is committed. Other actions still update and process the object's data; the only restriction is on the precommit procedure. Although this restriction may seem to create potential bottlenecks, the simplifications it provides in the processing of commits will keep the blocking intervals short enough so as to cause no problems. In particular, this restriction means that the storage manager must provide reliable updates for only one action per object per time period.

There are two levels at which the storage manager must supply this sort of reliability: at the partition level, and at the segment level. The partition has critical data which must be updated correctly to allow the storage manager to function correctly. As stated previously, this data includes the partition directory and the partition page map. At the segment level the storage manager is responsible for the consistent update of object data and the underlying structures that represent this data. We use two rather distinct approaches to providing the recovery for these two levels. In both cases the techniques provide pessimistic recovery; no changes are actually made to the "live" data until the responsible action commits.

8. The programmer may define other forms of synchronization within the implementation of the object based upon semantic knowledge and other design factors. The kernel does not preclude such choices.

7.1 Segment level recovery

Segment recovery is accomplished via a shadowing scheme^[11]. That is, segments on which actions are operating will have shadow versions which the actions will actually see. We note that one of the goals of the recovery scheme is, aside from producing consistent results, to allow recovery of segments (and partition structures) with as little storage overhead as possible, and with as few storage accesses as possible. Shadowing, then, will be minimal, with only those parts of the segment actually modified being shadowed.

The shadowing scheme consists of a set of protocols that indicate what the storage manager must do for specified segment states and action events. We consider these states and events in the following paragraphs and develop the protocols that shadow segments. When an action is started, the storage manager is involved initially in the transfer of the data for the object being operated upon from storage to memory. Until precommit occurs, the only transfer of information is from device to system. All modifications to the action data are handled in memory by the action manager. On the action commit the storage manager starts transferring information back to storage. These transfers are the result of the action management system protocols for transferring action updates to the permanent state of the object.

7.1.1 The precommit protocol

The precommit protocol ensures that updated pages of object data that an action has modified are recorded on non-volatile storage to prepare for the final commit of the action. The storage manager performs the shadowing and data transfers as follows:

- P1 The storage manager determines how many pages are to be shadowed and allocates storage for shadow versions through calls to the virtual memory system and the partition object, respectively. The storage manager allocates shadow storage not only for modified data pages, but also for the segment header, plus any index pages that are required to reach a modified data page.
- P2 The storage manager shadows the segment. The segment header is copied to the shadow segment header. The modified data pages are copied from memory to the shadow data pages. Modified versions of index pages are copied to shadow index pages. Some index pages must be modified and shadowed so that the shadows point to the shadow versions of data pages. The storage manager places a modified version of the segment header into the shadow segment header. Modifications made to the segment header data could include a change in the size, and changes to the array of pointers (some of these pointers may point to shadow pages, as with the index pages).
- P3 The permanent segment header is modified so that the status flag indicates that the segment is being shadowed. A pointer is also set in the header which indicates the location of the shadow segment header.

One point to note about the above protocol is that there are a number of reads assumed to get the segment structure into memory. Also note that the number of pages that must be shadowed and the identification of which index pages must be shadowed can be determined by knowing the size of the segment and which data pages must be shadowed. The segment header is modified last to reduce the work necessary to restore the segment in the event the system crashes before the precommit is completed.⁹

Once the precommit completes, we are left with two versions of the segment. The two versions overlap in spots as illustrated in Figure 8, where blocks within the dashed box are part of the

9. A crash at any point before this final write will recover with the shadow pages still listed in the free space list and completely unreferenced, and thus they get scavenged automatically.

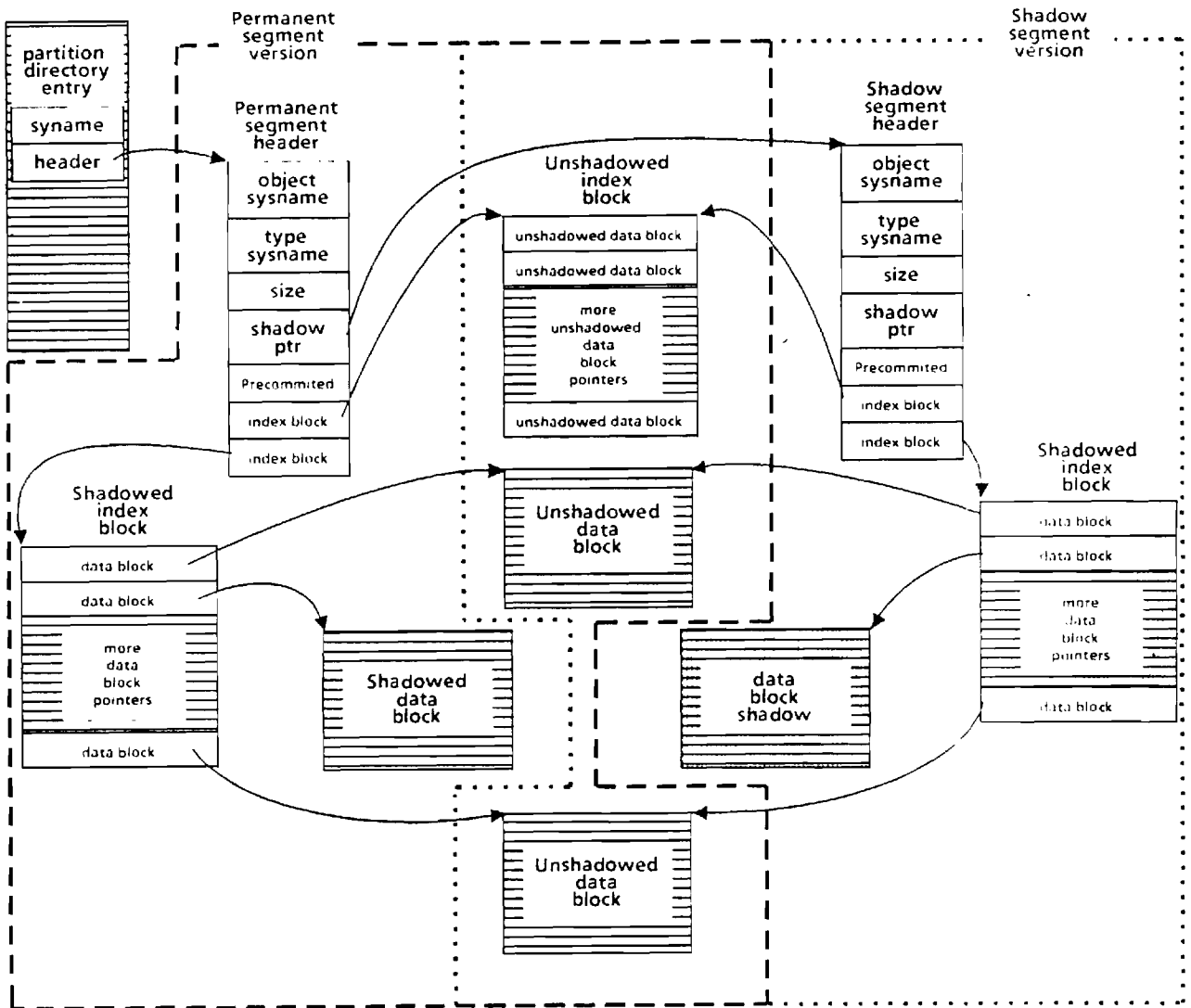


Figure 8. Precommitted segment

permanent version, while blocks inside the dotted box are part of the segment shadow. Read operations on unshadowed pages refer to permanent pages. The shadow version is visible only to the action which is performing the commit.

We must point out that the storage manager's precommit protocol is not the same as the action manager's precommit. After the storage manager has completed the shadowing, the action could still abort and the shadowed version would have to be removed. An example of such a situation is when the action spans several nodes and uses a two-phase commit protocol. Phase one is complete only when all nodes have completely shadowed any object data the action touched on their storage. If one node cannot do this, the action aborts.

7.1.2 The commit protocol

Once the segment is shadowed and the action decides that it can continue the commit, the storage manager performs its own commit protocol. The storage manager must switch the shadow version for the old permanent version of the segment. There is some bookkeeping for the partition as well. The protocol is as follows:

- C1 Update the permanent page map on storage. This requires that all addresses for shadow records be allocated in the page map and all modified records of the segment including the segment header be deallocated in the page map.
- C2 The partition directory is set so that it points to the new segment header for the segment.
- C3 The shadow segment header is set so that it is now the permanent segment header, that is, it is marked as "permanent."

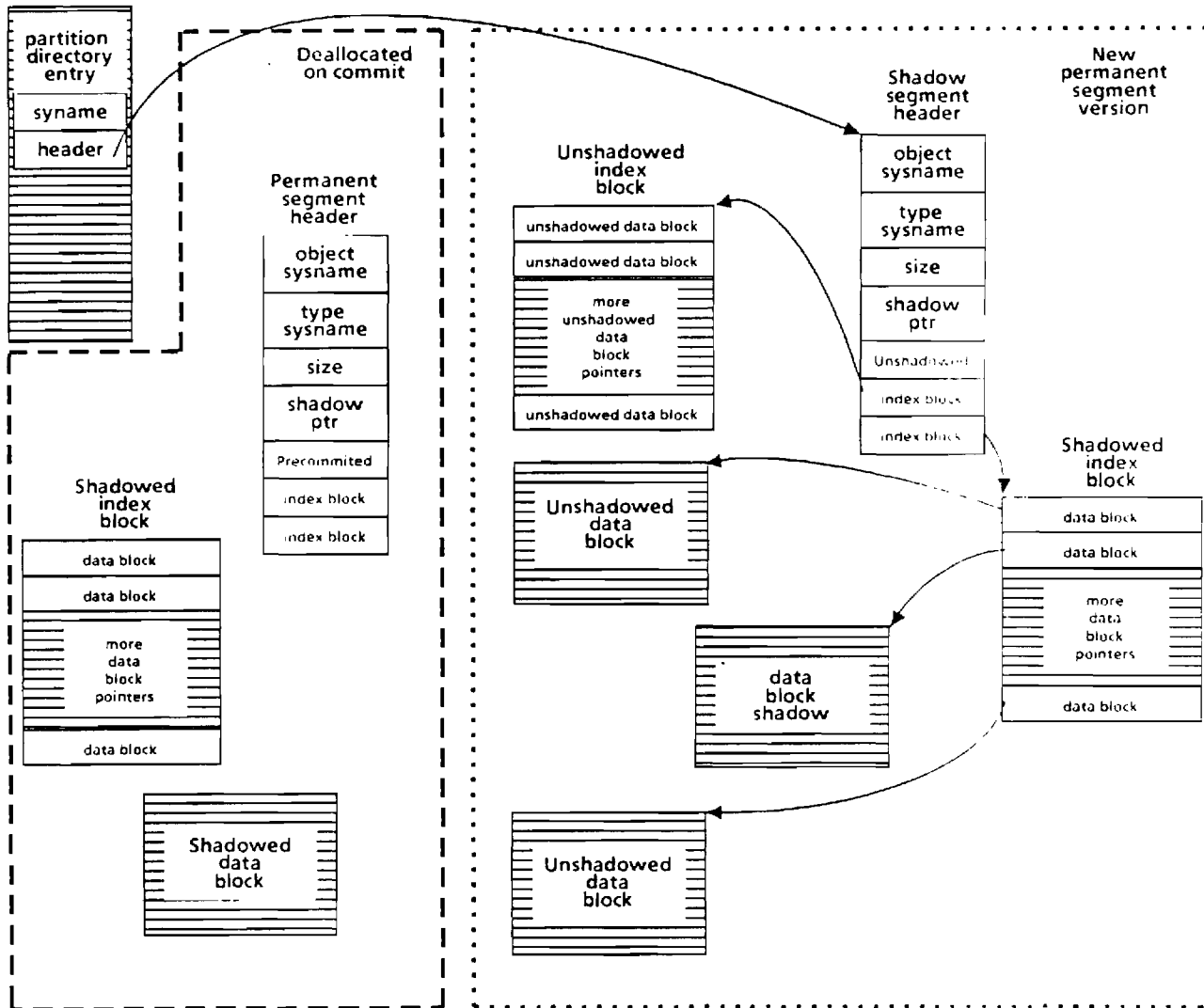


Figure 9. A committed segment

Once this protocol is complete, any references to the segment will refer to the new version of the segment. The new segment is a merging of old unmodified records and new records. Figure 9 shows a committed segment. The blocks in the dashed box were parts of the permanent segment being shadowed during precommit. These blocks are deallocated as part of the commit during step C1. During this phase of the protocol, the storage manager updates the permanent page map on secondary storage. Recall that Clouds uses pessimistic recovery and any effects of an action, including storage allocation to perform the commit, cannot become permanent until the action commits. Therefore, all allocations are performed on a volatile page map. We discuss this and other ideas in the section on partition level recovery.

7.1.3 The abort protocol

Actions can also abort for one reason or another and the storage manager requires a protocol for this event as well. The protocol simply rids the segment of any trace of the action's work as follows:

- A1 The volatile page map is updated to remove allocations that the action has made to shadow the modified pages of the segment.
- A2 The status flag of the permanent segment header is set to show that the segment is unshadowed and then the shadow pointer is set to null.

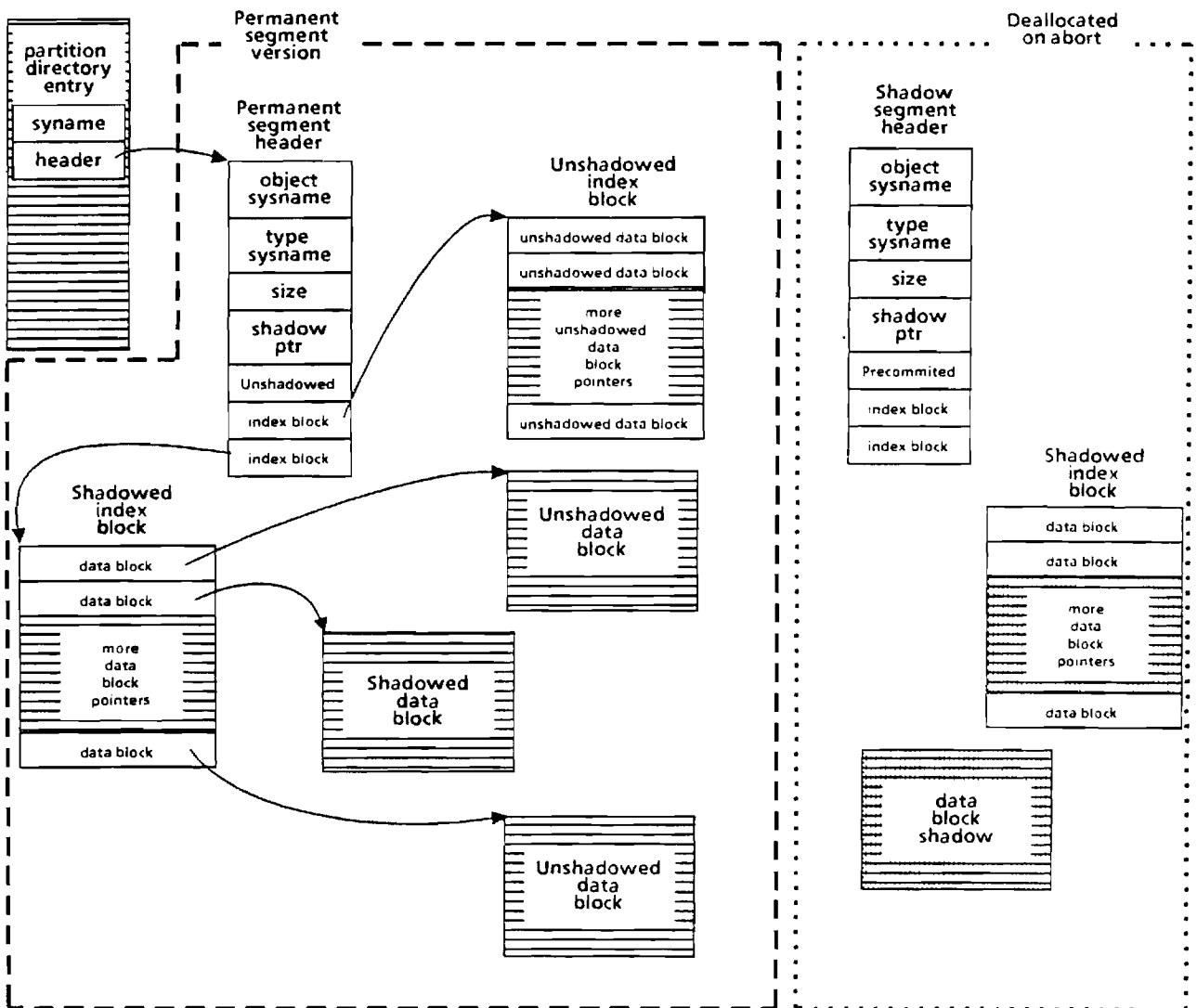


Figure 10. An aborted segment

The storage manager uses this protocol only when an action has started to commit and aborts in the middle. If the action aborts before attempting to commit, the storage manager is not involved at all. Figure 10 illustrates the results of the abort protocol. In this case, the blocks inside the dotted line are deallocated upon the abort, as these blocks are only shadows for the permanent segment.

7.1.4 System failures

One final event must be considered. That is how does the system recover from a machine crash? Specifically, we are concerned with restoring the segment and partition to a consistent state after the system is brought up again. The system may have had a number of actions in various states at the time of the crash and we want to insure the appearance of indivisibility of actions. Under the Clouds policy, any action that has not precommitted when a crash occurs is aborted when the system is restored. As we have already noted, actions which do not begin precommit before the system crashes do not concern the storage manager; these actions have no effect on system storage. For objects which completed precommit processing, we must determine whether their action's effects become permanent or are erased. This depends on the state of the action. The crash recovery protocol, then, is as follows:

CR1 A new volatile page map is created for the partition.

CR2 The storage manager determines which actions touched segments on this partition and determines the state of each such action. The storage manager polls a kernel database and examines the segments on its local storage to identify these segments.

CR3 If a segment was touched by an action that has completed phase one and should be committed, the storage manager performs the commit protocol on the segment, as above.

CR4 If the action which modified this segment was aborted by the action manager, the storage manager uses the abort protocol, as given above.

At the end of crash recovery, the partitions are in a consistent state; either the actions occurred or they did not. The database referred to in step CR2 is a kernel level database shared by the nodes in the system. The database exchanges information amongst the systems using a suite of algorithms developed in ^[1]. The information in the database represents an approximate state of the network. This database is copied from other nodes by the kernel when a node is added to the Clouds system. Among the information kept in the database is a list of actions, their status, and segments touched by the actions. Generally, the storage manager can find here the information needed for crash recovery. In some cases, though, a local action (one which does not leave the site on which it is born) may not appear in this list, even though its status at the crash time was complete and known. In cases such as these, the storage manager can find shadowed segments only by an exhaustive examination of the partitions.

Another issue is that of a system failure during an action write, so that only part of the write is actually completed. In the discussion thus far, we are assuming that we have atomic single record writes. The atomicity we are concerned with is failure atomicity, whereby the write either takes place or not. In practice, this means that we can detect an incomplete write (the system failed during a record write) and we are not overwriting the only copy of the data in question. If a device we are using does not support detection of incomplete writes, we can simulate the effect using the standard method of stable storage as described by Lampson and Sturgis in ^[12]. In ^[13] the question of when the atomic single record write assumption can be relaxed, if at all, and under what circumstances, is investigated.

7.2 Partition level recovery

In the last section we outlined the techniques used to provide reliability for the segments on storage. We now turn to the problem of maintaining the consistency of partition structures, particularly the page map and the segment directory. These structures were discussed to a small extent in the last section because they are involved in shadowing segments. We did not discuss how the structures themselves must be modified to maintain their consistency. Once again, let us consider the action environment provided by the kernel. Recall that a committing action blocks all other actions from committing in a segment it has modified. The partitions are objects, so that any action committing would block all other actions from committing in any object residing in that partition. For a one partition node, this permits only one action at a time to commit. We feel that this is too restrictive.

We allow any number of actions in a partition to commit simultaneously, excluding any segment conflicts. Given this, we do not feel that shadowing can be used to provide recoverability of the page map and directories. Maintaining the various shadow versions in itself would be complicated, but in addition we would need to propagate committed data to as yet uncommitted shadowed data. We therefore reject our segment level shadowing scheme as an approach for partition level recovery and we must develop another method for this task.

The partition directory does not have a volatile component. There are two copies of the directory residing on the partition (for the redundancy necessary to protect against media failures) and a committing action on a partition object must update both copies in a consistent manner to indicate that the new object version is to be used. Once again, we assume atomic single record writes, which will allow us to determine whether the copies are consistent, when the writes are performed in a determined order. An examination of both permanent copies and the header of the segment involved, if done in the proper order, will reveal any inconsistencies and the manner in which they should be resolved.

The partition page map has a volatile component which the storage manager uses to make non-committed storage allocations and which disappears after a system crash. Note that the volatile page map provides correct storage allocation information excluding system failures. Now recall that the commit protocol for storage management entails three steps, the second of which involves installing the action's storage allocations onto the permanent page map. We have two approaches we feel will provide consistent updating of the permanent page map. The first approach simply does away with the permanent page map of the partition, and maintains only the volatile version. As noted earlier, this provides correct storage allocation until a system failure occurs and the page map is lost. Clearly, we must be able to recover the page map after the system is restarted, and the obvious solution is an examination of the partition. Equally clearly, this will require quite extensive processing upon system startups.

The second approach to maintaining the partition page maps involves the use of intention lists and does require a permanent copy of the page map. With this approach, the storage manager during step one of the segment commit protocol does not write directly to the permanent page map, but instead writes an intention list of storage allocations (deallocations) to disk. Because the volatile page map reflects the correct storage allocation for a partition, the actual updating of the permanent page map from the intention list can be performed as background processing by the storage manager. If the system crashes before some updates are performed, they can always be done as part of the system startup processing. The steps required by this protocol are shown below:

1. The creation of the intention list begins at precommit. When the shadow is allocated, the storage manager places these pages on the allocation intention list. The pages to be replaced by the shadows are placed on a deallocation intention list.
2. When the signal is given to start the final commit, these lists are written to a list of pending allocations maintained by the partition.
3. At some later time, these lists are merged into the page map as part of normal partition bookkeeping.

The only restriction is that the updates from the intention list must be performed in the order in which the allocations and deallocations were committed.

Our initial implementation of the storage manager will use the first mechanism. We have two reasons for doing this. First, we are concerned more with the cost of commit processing than we are with system startup processing simply because we feel that system failures will be infrequent and because action processing is our model of computation. This approach both simplifies the implementation and makes the commit process more efficient, since no extra disk writes are required to update a permanent page map.

Secondly, an extensive examination generally will be made of the partitions at system startup to clean up any unfinished action commits or aborts. The reconstruction of the page map is partially subsumed in this processing.

7.3 Device support for recovery

The above protocols have several implicit assumptions on which they rely to operate correctly, two of which concern the device object. We have already mentioned the assumption that devices can perform atomic single record writes. The other assumption concerns the transfer of data from system to storage. The protocols assume that upon completion of a call to any of the "write" operations the data intended for transfer to storage has, in fact, been transferred. Under conventional systems, this is not necessarily the case, since requests for writes to storage may be buffered. Data may or may not actually be transferred before the system crashes. If the data were not actually transferred, there is no way to recover the segment or partition when the system is restarted.

At the device level, then, the storage manager requires some way in which to ensure the timely completion of data transfers. We wish to accomplish this without adversely affecting the other processing on the system. Also, the action causing the writes to storage must be informed of the completion of the writes in order to continue its commit processing.

There is a great deal of latitude with the timing of when the action writes are forced to the device. One discipline is to have a synchronous write operation that immediately forces the device to schedule requests issued by the operation. By this we mean that any requests currently being processed are completed and then normal scheduling is pre-empted. Synchronous write requests are then carried out in order of receipt. Thus, action writes are forced to the device early in the sequence of action commit processing. The drawback is that requests for synchronous writes appear in bursts at precommit and commit. Any scheduling that the device does for efficiency of the device's operation is disrupted.

Another approach is to allow the device to schedule the requests subject to its own constraints and simply inform the storage manager when the requests are completed. This allows the devices to schedule requests efficiently, but can delay action commit processing. However, the storage manager does know when the completion of the precommit and commit protocols can be safely signalled.

A compromise approach initially allows precommit and commit to be enqueued as usual and handled as normal requests. It is only when completion of the commit or precommit is imminent that the write must be forced to storage. To accomplish this, requests must be identifiable by the storage manager so that the manager can signal which requests must have priority. The manager can simply place the action id of the committing action in a field of the request when requesting a write to storage.

When the storage manager determines it is necessary, it can make a call on the device object to reorder its queue of requests, giving priority to this action's requests. This technique may prove useful if a significant amount of time can elapse before the storage manager must complete the precommit and commit procedures. In cases where the action has touched a number of objects on several systems this may indeed be the case. In such situations, the devices can operate efficiently (and possibly reduce the number of pending precommit and commit requests, reducing the disruption when it becomes necessary to force them to storage), and the action is not delayed, since it is not ready to complete its commit. To accomplish this as stated, the storage manager must be able to identify when requests must be forced to storage. This will be based on the results of any two phase commit that is performed and the storage manager will rely on the action management system to signal when final commit is to be performed.

Each device object maintains a flush table (as discussed in section 4) to control the forcing of action writes. When the list of requests for the action entry in the flush table is empty, the storage manager can inform the action that the commit processing can continue.

7.4 Summary

Support for reliability and recovery is integrated throughout the storage manager from the lowest level to the highest. The segment system, via the use of segment objects, provides for recovery of client object data recovery through the use of shadowing of modified data and the discipline of the shadowing provided by the protocols discussed above. The data that the storage manager uses to manage Clouds objects is made recoverable by the partition objects. At this level, our primary concern is how to maintain the data across system failures, and we present a few approaches for doing this. At the device level, support is provided to ensure that data is written when necessary, allowing action processing to be performed correctly at a higher level.

8. Conclusions

The motivation behind the Clouds project is the belief that systems in general and distributed systems in particular should provide reliable data management and reliable computation. This report documents part of our efforts towards that goal, namely the storage manager for the Clouds kernel. The Clouds storage manager, in addition to providing the traditional services of storage management, also provides support for the object-action methodology presented by the Clouds kernel.

We have presented an overview of the storage manager for the Clouds kernel. The storage manager is presented as a collection of objects, each of which provides an abstract view of the secondary storage. At the lowest level, secondary storage is viewed through the device object, and the physical storage medium is viewed as a sequence of pages (in the current implementation, a page is 512 bytes) with very little structure, other than the device header and index table. One step higher in our hierarchy is the partition object, which manages a portion of the raw storage provided by the device object. Once again storage is viewed as a sequence of pages, but that storage has a more defined structure. Each partition maintains a directory and a page map, so that each partition is responsible for managing its storage and for providing a location service for the next level of abstraction, the segment object. The segment object provides a view of storage that is a sequence of bytes and each segment object generally corresponds to some other kernel or user object. The storage manager views segments as a tree-like structure of pages.

We have described the data structures associated with each object and presented the operations with which the data structures can be manipulated. We have also tried to convey the relationships amongst the three objects and to show how they interact with each other and the rest of the kernel.

The research that we are conducting is primarily involved with how the storage manager provides the recoverability of the storage it manages and thus supports the reliability of the Clouds kernel. To that end the storage manager uses a set of protocols to ensure that object data is updated in a consistent manner and that even through system failures, enough information survives to maintain the consistency of the object. We show how these protocols are used to support the action/object programming paradigm of the Clouds system.

Each level of storage object discussed provides some support for recoverability. The device objects maintain flush tables which allow the storage manager to ensure that action writes are completed before a commit is finalized. The partition object maintains a consistent view of allocated storage and insures the correct updating of the partition directory. The segment object provides recovery of object data through the set of protocols described.

REFERENCES

1. Allchin, Jim, *An Architecture for Reliable Decentralized Systems*, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1983.
2. McKendry, Martin, *Ordering Actions for Visibility*, Technical Report GIT-ICS-84/05, Georgia Institute of Technology, Atlanta, Georgia, 1984.
3. Allchin, Jim and Martin McKendry, *Object-Based Synchronization and Recovery*, Technical Report GIT-ICS-82/15, Georgia Institute of Technology, Atlanta, Georgia, 1982.
4. Allchin, Jim and Martin McKendry, "Synchronization and Recovery of Actions," *Proceedings of the Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, 1983.
5. Spafford, Eugene and Martin McKendry, *Kernel Structures for Clouds*, Technical Report GIT-ICS-84/09, Georgia Institute of Technology, Atlanta, Georgia, 1984.
6. Spafford, Eugene, *Kernel Structures for a Distributed Operating System*, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, in preparation.
7. Jones, A. K., "The Object Model: A Conceptual Tool for Structuring Software," *Operating Systems: An Advanced Course*, Springer-Verlag, New York, pp. 7-16, 1979.
8. Wilkes, C. Thomas, *Preliminary Aeolus Reference Manual*, Technical Report GIT-ICS-85/07, Georgia Institute of Technology, Atlanta, Georgia, 1985.
9. LeBlanc, Richard J. and C. Thomas Wilkes, "Systems Programming with Objects and Actions," *Proceedings of the Fifth International Conference on Distributed Computing*, Denver, 1984.
10. Bloom, B. H., "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, No. 13, Vol. 7, pp. 422-426, July 1970.
11. Gray, J. N., "Notes on Data Base Operating Systems," *Operating Systems: An Advanced Course*, Ed. by R. Bayer, R. M. Graham, and G. Seegmuller, Springer-Verlag, Berlin, 393-481, 1979.
12. Lampson, B. W. and H. E. Sturgis, *Crash Recovery in a Distributed Storage System*, unpublished paper, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California, 1979.
13. Pitts, David V., *Storage Management for a Reliable Decentralized Operating System*, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, Georgia, in preparation.

Kernel Structures for Clouds *

Technical Report
GIT-ICS-84/09

March 1984
Last Revision: May 14, 1984

Eugene H. Spafford
Martin S. McKendry

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

* This research is funded in part by NASA grant NAG-1-430
and by NSF grant DCR-8316590

1. Introduction

In the past few years, a great deal of research has been focused on the potential benefits of distributed systems. In particular, a distributed system offers the potential of a fault-tolerant computing environment. A distributed system also suggests increased computing power through the combination and application of resources. The presence of multiple machines, however, raises many questions relating to communication, consistency, reliability, configuration, and user interfaces, to name just a few. These questions are difficult to address, and that is perhaps the reason why so few attempts have been made to construct actual distributed systems. Interesting recent work in this area includes the *Eden* project at the University of Washington (e.g., [Alme83]), the *Argus* project at MIT (e.g., [Lisk83] and [Weih83]), the *Accent* system at CMU ([Rash81]), and the *ISIS* project at Cornell ([Birm84]).

The *Clouds* project is an approach to the construction and application of a distributed system that is intended to address these questions. We support the "room full of computers" view of distribution. In this view, the user sees a single resource, despite physical distinctions. In our research approach, this is achieved by constructing a highly transparent multicomputer operating system with low-level support for maintaining consistent data items. A *multicomputer* or *computer cluster* is a system of many computers joined into one large system. The system's distribution is *transparent* to users and to most operating system components in the sense that the user is not aware of the nature or number of machines which compose the multicomputer. The user's data and processes may be distributed throughout the multicomputer system, or they all may be located on one processor -- there is no observable difference to the user, nor is there any need for the user to be aware of the configuration. We support this transparency during *upward configuration* -- the addition of more machines, and during *downward reconfiguration* -- the removal or failure of machines.

Clouds supports abstract data objects at a very low level. These objects are used to build the operating system and applications. Some of these objects may be made *recoverable* (operations on those objects may be undone or reversed in the event of failure or error). *Atomic transactions* or *actions* are used by both the operating system and user applications to maintain consistency and recoverability of data and operations. The design makes use of actions and objects to provide reliable operating system services, such as job schedulers, and thus provide a fault-tolerant system.

The principles and motivations behind the *Clouds* project have been described in more depth in several documents ([McKe83], [McKe84], [Allc83a]). The authors assume that the reader is already acquainted with the *Clouds* project and is somewhat familiar with the goals outlined in those documents. This paper is intended to be an introduction to the internal structures of the *Clouds* kernel. We will be constructing an experimental *Clouds* system during the next few years using dedicated minicomputers and personal computers. Further description of the *Clouds* kernel will be done as this experimental system continues to be designed and constructed ([Spaf84], [Spaf85], [Pitt85]).

2. Basic Assumptions

2.1 Terminology and Logical Structure

The term *computer* is used in this paper to mean a physically-discrete computer. Each computer supports an instance of the *Clouds sub-kernel*. The sub-kernels implement the *Clouds* virtual machine on each computer. The copies of the sub-kernels communicate with each other and together

form the *kernel* for the system. The operating system itself is implemented above the kernel, and applications are programmed above the operating system.

Figure 2.1 illustrates this logical hierarchy of levels.

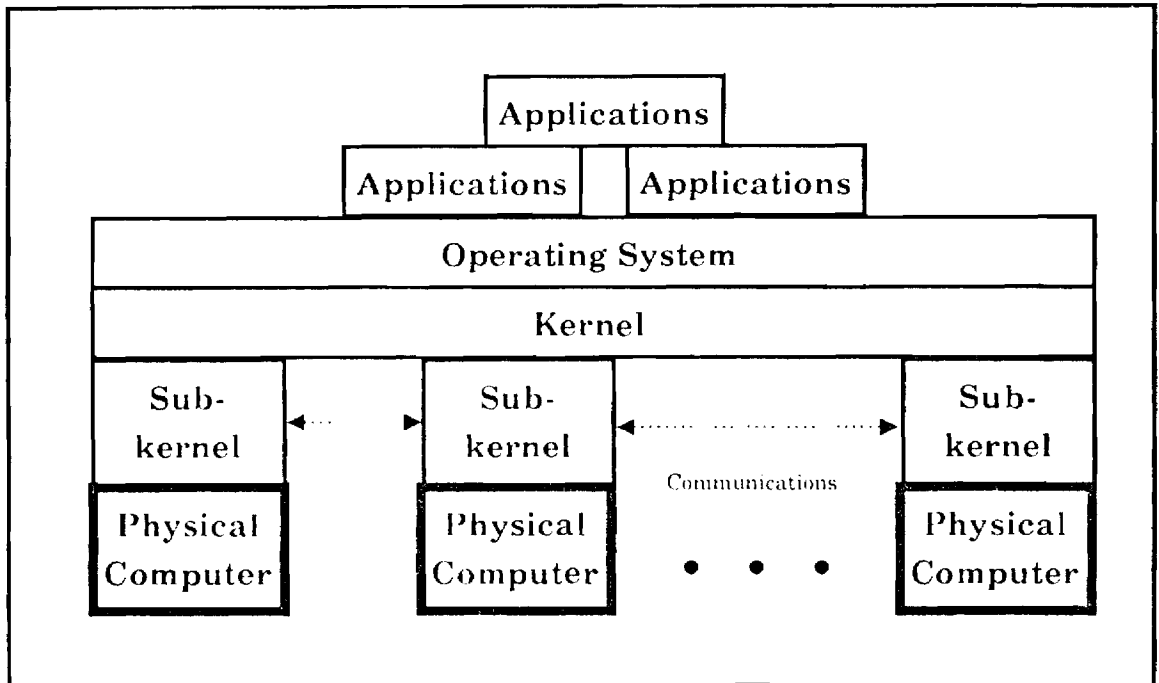


Figure 2.1 -- Clouds logical hierarchy of levels

2.2 Hardware Structure

The *Clouds* multicomputer is composed of a number of minicomputers connected by one or more communications paths, and accessed by "intelligent" terminals. The minicomputers are likely to be in close physical proximity, while the terminals may be somewhat more distant, but will still be within 1-2 km. Although isolation of a single machine is possible (a *trivial partition*), we anticipate that the probability of a general partition (disruption of communication so as to form functioning, but isolated groups of processors) will be small.

Our first prototype system will consist of three or four Vax 11/750 processors connected together by a fault tolerant 70Mb/sec bus. These systems will also be connected by a 10Mb/sec Ethernet, and possibly through dual-ported disks. A number of IBM PC microcomputers will also be connected to the Ethernet and will serve as intelligent terminals. Figure 2.2 illustrates the connections.

2.3 Access rights, names, and capabilities

The system references objects with unique identifiers. Each item is referenced uniquely via a 32 bit quantity known as a *sysname*. The *sysnames* are unique in time and space -- non-identical items have different names. *Sysnames* are composed of a node ID and a sequence number, which together form the *birthmark*. The sequence number is composed of two fields -- the *subsequence number*, and the *crash count*. The *subsequence* field is incremented for each new name request. The *crash count* is incremented each time a node is restarted, and each time that the *subsequence* field overflows. The

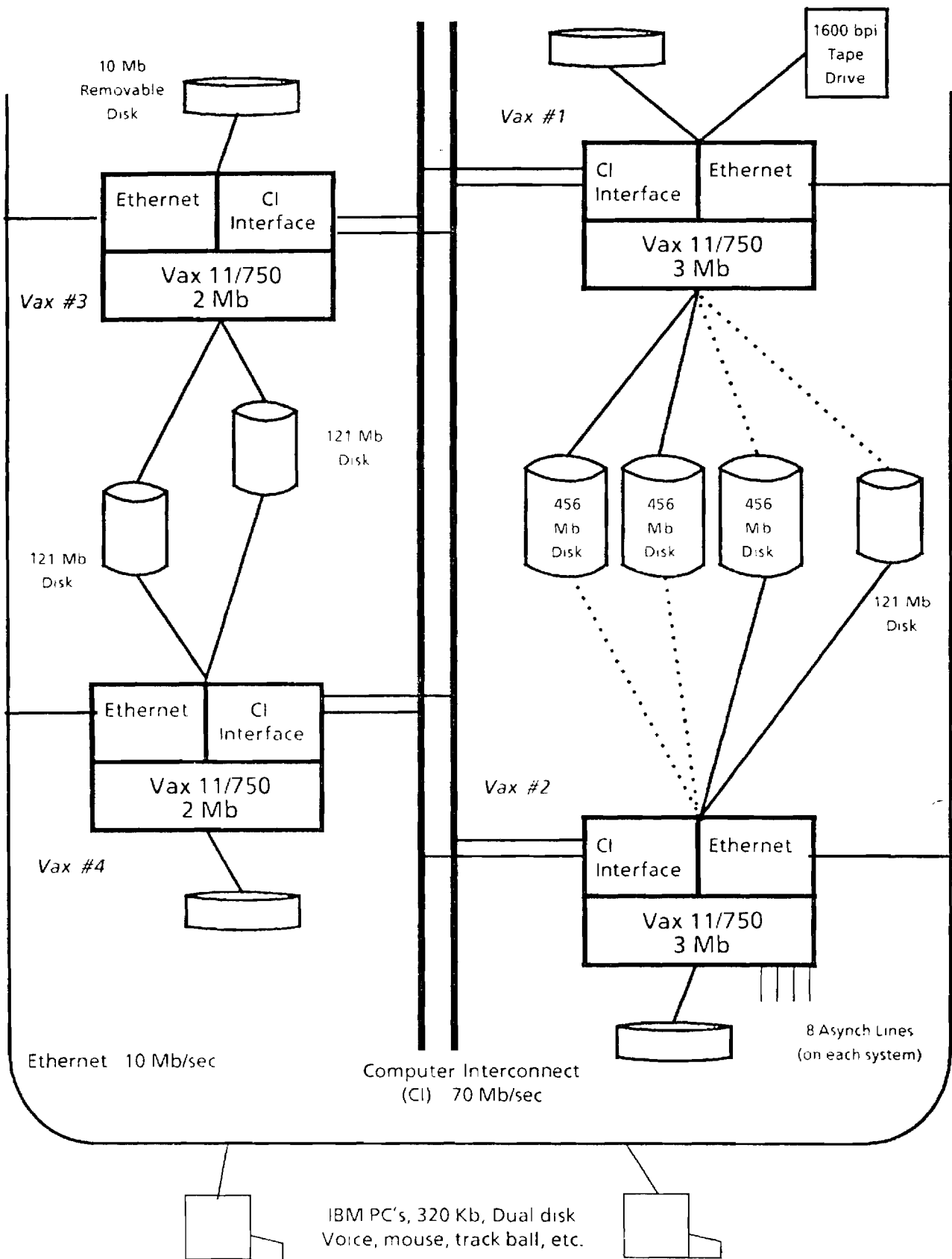


Figure 2.2 -- Prototype Configuration

sysname also contains a field which helps determine the type of the item referenced by the sysname -- a system procedure, a user-defined object (referred to as a *client object*), a process, and so on.

Objects are referenced via object capabilities. An object capability is a 64 bit value consisting of the sysname of the object instance being referenced, and a 32 bit capability mask defining the access rights to the object. Each bit set in the mask indicates an operation that can be executed by the holder of the capability (those operations being present in the object). (Refer to figure 2.3.) Items being referenced by the system have *implied* access rights for certain kernel operations. These *implied* rights always allow the kernel to invoke the operations, but the ability to invoke those operations cannot be passed to user processes.

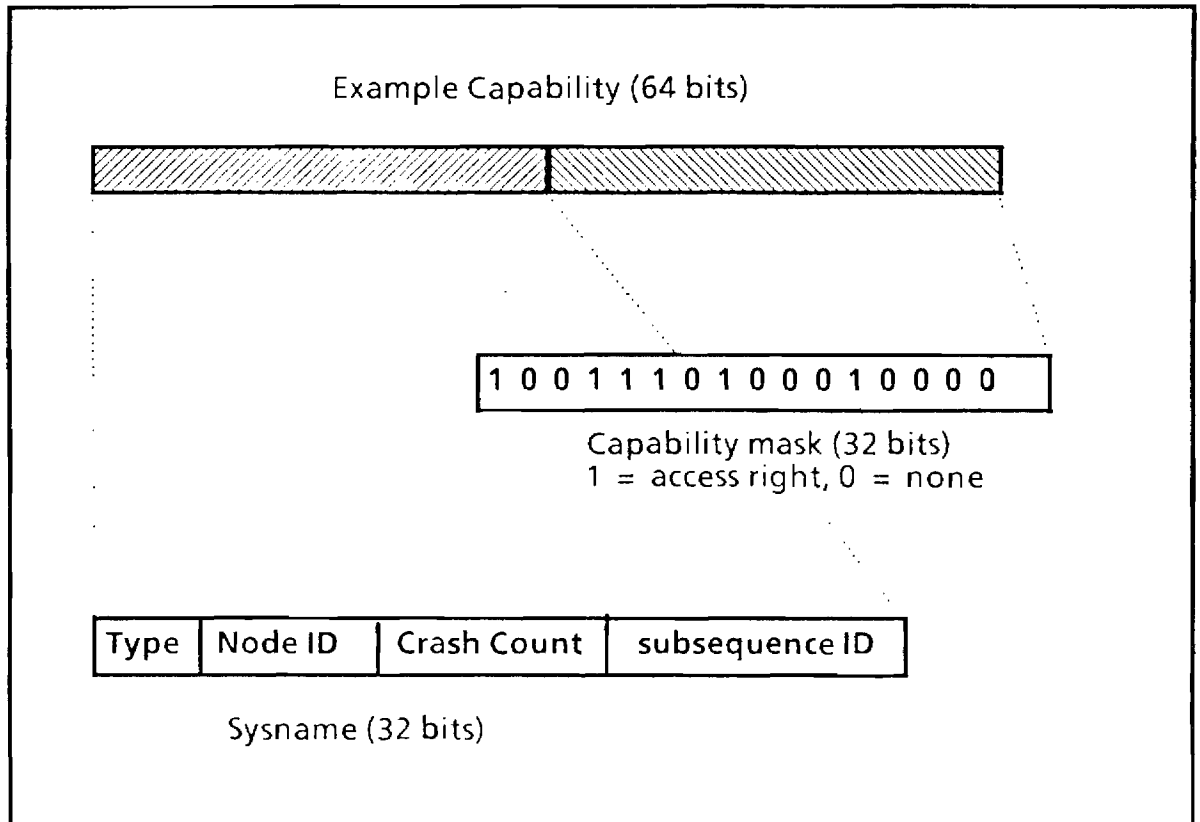


Figure 2.3 -- Capability structure

The existence of objects implies other capabilities. Any reference to an object also implies a reference to the type manager for the object (described in the next section), and to the system object manager. The kernel has an implied capability consisting of the sysname and all access rights. This implied capability may be used to invoke certain operations (e.g., abort, initialize), and it may be used when managing the segment in which an object resides. These implied capabilities and references are all used internally by the kernel and their existence is not seen by application software.

3. Objects

Objects are the fundamental data abstraction in Clouds. The rationale behind their use, and specifically their use in Clouds, has been described in other papers (e.g., [Allc83a], [Allc83b],

Kernel Structures For Clouds

[McKe83], [Jone79]). Objects in Clouds are *passive* unlike Objects in Eden [Alme83]; there are no processes resident inside the object. This section describes the structure of objects and the manner in which they are created, deleted, and in which operations are invoked.

3.1 Structure (types and managers)

Objects can be viewed as consisting of two parts: an object type manager and an object instance (the "type" and "instance," respectively). The type manager consists of procedure code which is allowed to manipulate the object during creation and deletion of the instance, a template of the uninitialized instance, and certain other bits of information used by the kernel in object management. The type manager operations are invoked to create and delete instances of the object type, move instances of the object from one location to another, modify existing instances, and other related operations.

The type managers are objects in their own right: their associated type manager is part of the kernel and is known as the *object manager*. There is an object manager in each sub-kernel which communicates with all of the other object managers in the system. Figure 3.1 illustrates the logical relationships amongst objects instances, object types, and object managers.

Object instances consist of data comprising the object, procedure code which operates on the data when operations on the object instance are invoked, access and modification information, synchronization variables (if appropriate), and other information related to the object instance. As an optimization, the procedure code for object instances can be stored in one single location (e.g., in the type manager) and shared by all of the active instances. The object can be thought of as composed of the code (which may include action-oriented operations such as *commit* and *abort*), permanent data, and volatile data (such as heaps or stacks) which disappears when the object is not in use (see figure 3.2).

Each instance and each type is stored as a segment (see section 6). When an operation is invoked on an object instance, the kernel maps the code for the type manager and the instance into the virtual address space of the invoking process. The object managers and segment system are responsible for finding objects when presented with the capability, and with mapping those objects into virtual memory space. This will be discussed in more depth in section 3.4.

3.2 Object Creation

To create a new object, the user first describes the object type using an appropriate applications programming language. This forms a template of the data structures which compose the data portion of the object. The user also codes functions which operate on this data. These functions import and export arguments by value only; reference parameters are simulated by passing capabilities by value.

The procedural part of the object definition may contain special routines for synchronization of access to the object, support of atomic actions, and initialization of new instances of the object. These routines may be derived from a standard system library, or the user may program them specifically. Every type manager contains functions for creating a new object instance, deleting an old object instance, and for initializing a new object instance. Each recoverable object must also contain functions to implement beginning of action (BOA), abort, precommit, and commit.

Once the object implementation is written, it is compiled by the appropriate system compiler(s). The result will be a file of code and data which is passed to the system object manager via a call through the kernel interface. The object manager will change the type of the file to "type manager" and will return an object capability to the new type manager.

To create an instance of an object, the user invokes the "create" operation on the type manager. The calling process also specifies a storage partition (cf) where the permanent version of the object

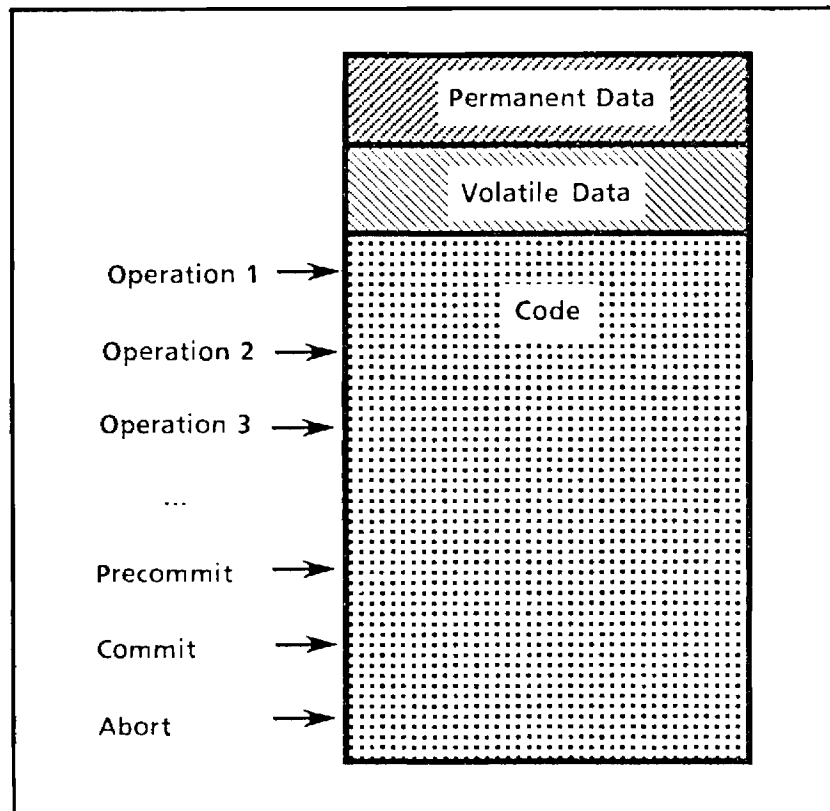


Figure 3.2 -- Picture of an Object

instance is to reside. The type manager uses this partition capability to create a new segment of type "client object." The sysname returned by the segment system is used by the type manager to create the capability returned to the user. The type manager also initializes the object capability mask so that all defined operations are enabled (e.g., if there are only 10 operations defined on the object, the remaining 22 bits in the mask are not set).

Once the type manager has created a segment, it uses its data template and "init" function to initialize the new object instance. This usually involves setting some data values to initial states, initializing the heap for the object, resetting and defining synchronization variables, and creating a skeleton VAM (Virtual Address Map -- see section 5) which will be used to map the object instance into a process's address space when needed.

When everything has been updated, the type manager returns the newly created capability to the caller. Future references to the object instance will all be through this capability. The type manager also increments an internal instance count which may later be used when deleting the type manager (deleting the type manager while there are outstanding instances results in those instances becoming "orphans" which cannot be used), the instance count is always an advisory value and no guarantee is made about its accuracy.

3.3 Deleting Objects

To delete an object instance, the user invokes the "delete" operation contained in the type manager for the object. This is an operation on the capability to the object instance. Each object contains a pointer to the type manager for the object instance, and this pointer is used to find the type manager and

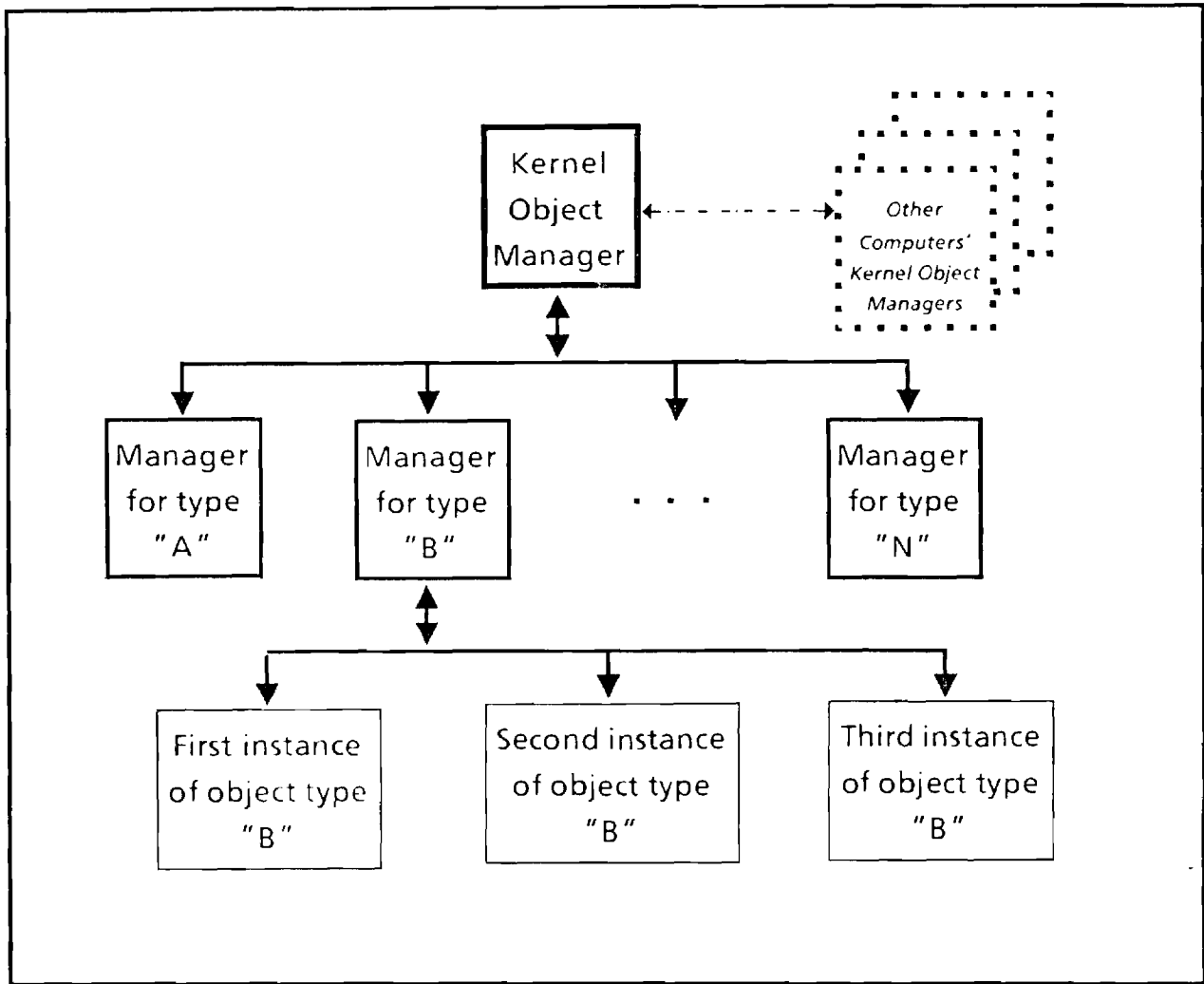


Figure 3.1 -- relationship of instances and types

invoke the delete operation, this is an example of the use of an "implied" capability. To delete an object instance the user must have the "delete" right, however.

Type managers are treated exactly like objects if the "delete" operation is performed on them. The system object manager acts as the type manager for all user-defined type managers, and it is responsible for performing the necessary delete operations. Note that the object manager has the option of checking the reference count inside the type manager and issuing some warning to the user if it is non-zero. That is, it is possible to advise the user if that type manager might still have outstanding instances in existence

3.4 Invoking operations on instances

An operation is invoked on an instance by making a call on the kernel with the capability to the object instance, the operation number, and (optionally) a list of parameters to the operation. The kernel object manager will first verify that the operation number is in range and that the capability mask

contains the required access rights. If the attempted access is not valid, the process or action attempting the operation is aborted (to prevent security violations and help contain errors).

Next, the segment system is presented with the object capability and requested to map the instance into memory. If the instance name is currently known to the segment system, then a mapping is already known. Otherwise, the segment system searches the directory of every local active partition. If the instance is found, then the mapping is performed and the segment is now "known" to the segment manager; future references to the segment will be to the segment on the local partition. This search mechanism is described in more detail in section 8 and in [Pitt85].

If the segment is not found locally, the segment system invokes a search module which attempts to locate a copy of the segment elsewhere in the system. If unsuccessful, then the current action is aborted since there is no way of deciding whether the desired object is locked, never existed, or simply cannot be found due to some failure in the system. If successful, the reference causes the object to be made "active" on the system where it is found. From this point on, references to the instance will continue to be mapped through the search module. Future references will be mapped to a system where the instance is active.

Once the segment containing the object instance is made active, the segment containing the object is read to find the name of the type manager for the instance. The type manager is then mapped into memory in exactly the same manner as the instance was mapped. The object instance may contain a "hint" which can be used to speed this mapping; a hint can indicate the site where the type manager was found when the object instance last referenced it.

After the instance has been mapped into the process's virtual address space, the operation is invoked with the appropriate parameters. Additionally, a capability to the object control block for the object is stored in the "current object" field of the process control block for use by the kernel, if necessary.

3.5 Cloning Objects

For enhanced availability and speed of access, it may be practical to have redundant copies of type managers located at different spots around the system. Thus, if a machine goes down and it makes a type manager inaccessible, it may be possible to map future references to a copy of the type manager located on a still-active machine; these copies are referred to as *clones*.

Currently, only immutable objects may be cloned. This means that only type managers may be cloned, since they contain unchanging code and data templates. The object manager has a *copy* operation which can be invoked to clone a type manager.

References to cloned type managers still occur through the standard capability mapping mechanism. All clones will have the exact same name and are completely interchangeable. Any reference can be mapped to any available type manager.

When a type manager is cloned, its reference count field is marked as "not valid" both in the original and in the clone. When a delete operation is performed on the type manager, one of the clones will be deleted. The application code may choose to warn the user that the type manager is a cloned object and there may be other copies in existence. The code may further note that there may be undeleted object instances which could be rendered useless by removal of the last clone.

4. Processes and Actions

4.1 Processes

The basic instrument of activity is the process. A Clouds process is similar to processes used in other systems. Each process represents an identifiable sequence of operations. Each process is represented by a unique *process control block* or *PCB*. The PCB for a process is used to store the contents of the machine registers when the process is not active. The PCB also holds pointers to the process's stacks, pointers to the structures currently defining the virtual memory space of the process, and a pointer to the object control block of the object that the process is currently accessing, if any.

A process is created when the kernel process manager receives a request for a new process. Each request specifies an activity that the newly-created process is to begin, and a pointer to an activity-specific block of parameters for the process to use. The process manager allocates space in its internal store for a new PCB and a new set of memory maps (see section 5 for more details). The PCB and memory maps are initialized, and the process stacks are also initialized. The system scheduler object is invoked with a capability to the process and the process is added to the ready list for subsequent dispatch and activation. A capability for the process is also returned to the activity which requested the creation of the process. That capability can be used to halt or kill the process, change its priority, or modify other operating characteristics of the process.

When a process has completed its assigned tasks it returns to the kernel process manager and its PCB and memory maps are reclaimed for use in building new processes. The process may also be halted and reclaimed by the process manager upon request. Such a request must contain a capability for the process and have the necessary access rights.

There is only one process running on a machine at any one time. The other processes in the system at that machine are either linked into the *ready list* or they are linked into the *wait list* associated with some synchronization item. The ready list is a list of processes awaiting a turn at the processor. The ready list is maintained in a scheduler object according to the scheduler's queuing algorithms. The process dispatching mechanism for the virtual machine invokes operations on this scheduler module to obtain the next ready process or to enqueue a process (make it ready). The scheduler object is not, strictly speaking, a part of the kernel. There may be one scheduler for many or all of the machines in a Clouds system, or there might be one per machine each using a different queuing discipline on their ready lists. This enables the system to change scheduler modules and disciplines without stopping so as to adjust to changing configurations and workloads.

The kernel supports traditional counting semaphores, single and multi-mode locks, event tickets, and timed events as means of synchronization. The process manager can create new instances of each of these items upon request. Each synchronization item is associated with a wait list that is a list of processes blocked on that variable. When a process blocks on one of these items it is removed from the ready list and linked into the end of the wait list associated with the item. A pointer to the synchronization variable is placed into the PCB of the process for later reference. When a process is unblocked, it is unlinked from the wait list and added to the ready list via an invocation of the scheduler.

4.2 Actions

Actions are sequences of operations which occur *atomically*. That is, to an outside observer, the operations performed by an action occur all at once or not at all. Actions in Clouds have been discussed extensively in [Alle83]. Briefly, in Clouds, actions are related to processes in that the operations associated with an action must be performed by one or more processes. The kernel object

manager is responsible for maintaining information about actions. It keeps a record of all objects visited by each action during the lifespan of the action. It handles the commit and abort protocols associated with actions. It is also responsible for ensuring that only processes operating in the context of actions are allowed to touch recoverable objects.

When the object manager receives a request for a new action it first obtains a new worker process from the process manager. It enters the sysname of the process into an *active action descriptor*, and assigns an *action name* to the process. That action name is copied into the PCB of the process, and is used to form a new capability to the action. This capability is then passed back to the requester. The requester is not given the capability for the process; all requests to the process manager are made on the action and through the object manager.

While an action is active, the object manager notes every recoverable object touched by the action and enters the sysnames of those objects into the action descriptor. Should the action abort itself or be aborted by some other process holding a capability to the action, the object manager will (eventually) invoke the "abort" operation on each recoverable object listed in the action descriptor. When the action attempts to precommit or commit, the object manager will invoke the corresponding operations on each of the objects listed in the action descriptor. Once an action aborts or commits, its action descriptor is deallocated and the process manager is called to reclaim any processes associated with the action; by definition, an action can perform no more operations after it has aborted or committed.

The synchronization of actions and multiple processes acting on behalf of the same action is left entirely to each object accessed and to the object manager. The kernel provides no direct support for deadlock handling. These can be resolved through the use of timeouts -- each request to create an action is accompanied by a value specifying a maximum completion time. If the action does not complete within that time interval then it is aborted by the object manager.

Clouds also supports *subactions*. These act like actions in virtually every respect except that their effects are not permanent until all of their top-level ancestor actions commit. An action is not allowed to commit until all of its subactions have completed by aborting or committing. When an action aborts, all of its subactions are also aborted. The algorithms defining the interaction of actions and subactions are described in [Alle83], as are the algorithms for dealing with various kinds of network and communications failures.

5. Virtual Memory

5.1 Mapping Virtual Memory

The virtual memory system maintains the binding of physical memory locations to references made by executing processes. In particular, we are interested in the mapping of locations inside *Clouds* objects in such a manner that we can invoke the operations on objects and modify object instances. Additionally, we wish to support shared objects, an embedded kernel, and efficient management of storage coupled with correct operation of the action/object mechanisms. These constraints have made a major impact on the form of the underlying virtual memory structure. The resultant structure bears some surface similarities to the virtual memory structure of systems such as HYDRA [Wulf74] and Accent [Rash81].

Before describing the structures used to support the virtual memory system, we present the address space as seen by a typical process, and describe the utilization of that space. The virtual address space can be thought of as composed of three portions -- object space, per-process space, and system kernel space (see figure 5.1). The object space contains the code and data associated with the currently active object; this may include executable code from the object type, data items and heaps associated with

Kernel Structures For Clouds

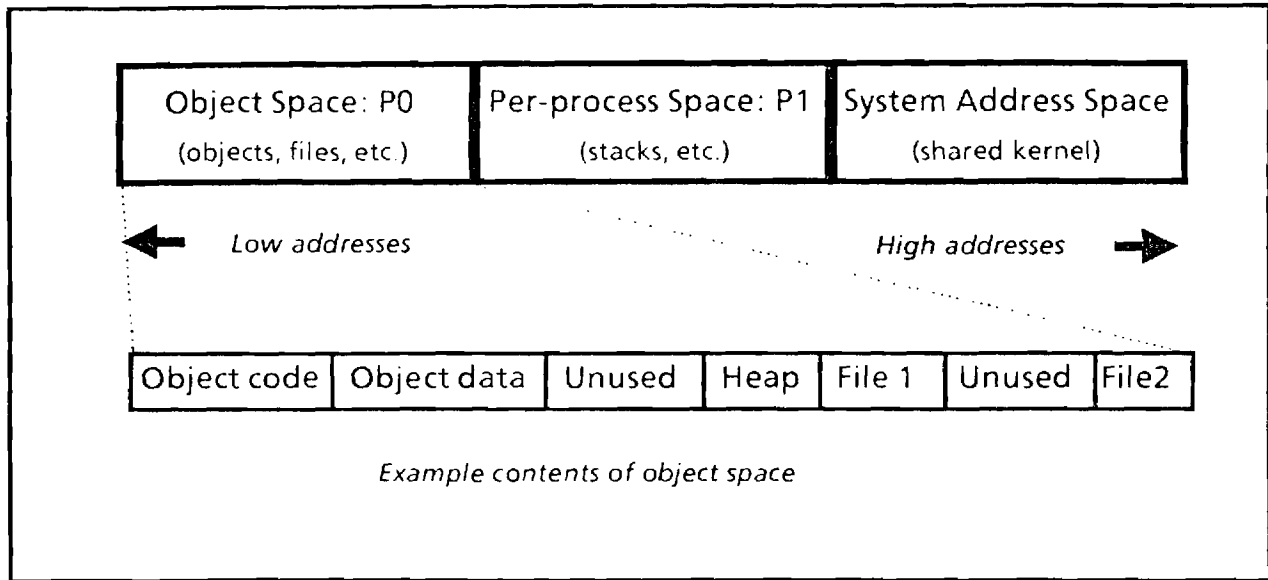


Figure 5.1 -- A process's view of its address space

the instance of the object, and one or more "windows" into files being referenced by the active object. On the Vax, this space corresponds to the P0 page map.

The per-process space contains items that survive object invocations and returns. That is, it contains items which are not specifically associated with any active objects but are instead associated with the active process. This space contains all of the user stacks and static data not associated with any object. On the Vax, this space corresponds to the P1 page map.

The system kernel space is the same for every process. Shared items such as the virtual memory tables and process control blocks reside in the system space. The system space is also where most of the code which implements the kernel resides. On the Vax, this corresponds to the area mapped by the system page map.

Each of these address spaces is mapped in a similar manner. Refer to figure 5.2 for the following discussion of common features

Contiguous, related addresses in a process' address space are mapped together as one "chunk." For example, all the executable code in an object type could be mapped as one chunk, as could an entire user stack, or the entire contents of a file. Each "chunk" is represented as an entry in a *Virtual Address Map* or VAM. There is one VAM describing system space, one unique VAM for each process' per-process space, and one VAM for each active object. These VAMs are referenced, respectively, by a pointer in the system control block, by a pointer in each PCB (process control block), and by a pointer in each OCB (object control block).

Each VAM has a header which points to a hardware page map. This pointer is loaded into the current process' page map register to effect the mapping indicated. That is, the page map table (PMT), contains the hardware defined entries necessary to define the virtual memory space in the physical page frames available.

The VAM header also contains a count (up to 8 in the initial implementation) of valid VAM entries following the header. Each VAM entry defines a "chunk" of the virtual address space. A VAM entry

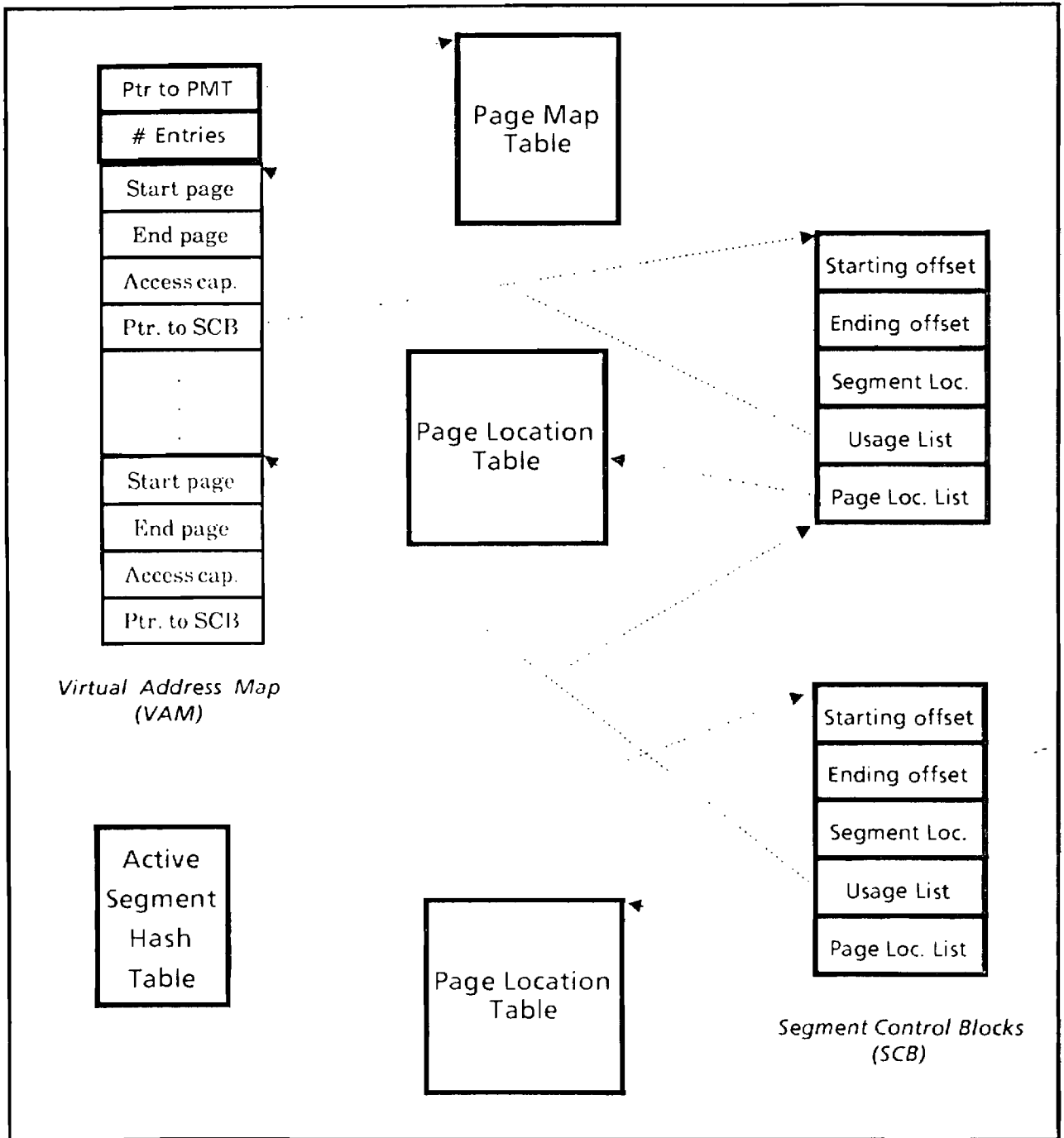


Figure 5.2 -- Common Features

contains the starting and ending page numbers of the process address space described by the entry. It also contains a pointer to a *segment control block* or *SCB*, and the capabilities that this process/object has in relation to that segment (e.g., read-only, read/write, delete, etc.).

Each SCB defines a currently active segment (segments are described more fully in the next section). The SCB indicates the portion of the real segment which is mapped. For instance, if the process is

Kernel Structures For Clouds

accessing only a few pages of a multi-megabyte file, there is no need to map the whole file segment into the address space. Instead, the SCB would contain the beginning and ending page numbers of the section that was being accessed (The VAM entry would indicate the addresses within the virtual address space where that segment would be mapped.)

The SCB also contains pointers to all VAM entries referencing the segment (the usage list), and a pointer to a *page location list* or *PLL*. There is one entry in the PLL for each page described by the SCB. Each PLL entry indicates the status of that segment page (one of: resident & locked, resident & unlocked, being brought in (in-transit), being removed (out-transit), on the pre-page list, non-resident and in the source partition image, non resident and on the paging partition, or not yet defined). The PLL entry also contains a location which is used in combination with the status to find non-resident pages and remove resident pages. The frame number of resident pages is also indicated in this list.

The relationship of these structures to one another may be made clearer by presenting a brief scenario of handling a page fault. Suppose an active process gets a page fault in its P0 (object) space. The page fault handling code locates the appropriate VAM by using the pointer located in the OCB of the currently active object (as noted above, a pointer to the current OCB is in the PCB of the current process). Next, the fault code compares the page number of the fault with the ranges presented in each VAM entry. When the appropriate entry is found, its pointer is used to locate the SCB which describes the missing page. The offset from the starting page number in the VAM entry determines the relative offset in the segment which is needed.

The pointer to the page location list in the SCB is used next. The fault code uses the relative offset to index into the PLL and obtain an entry corresponding to the missing page. Further action is determined by the current status of the missing page:

the page is resident. This implies that the page was being brought in at the time of the fault and has since arrived, or else the page is also in use by some other process and the presence of the page has not yet been indicated in this process's page map table. In either case, the frame number is taken from the PLL and inserted into the proper place in the PMT pointed to by the VAM. -

the page is undefined. This implies uninitialized data space, such as in a heap or stack. An empty frame is filled with zeros (for error containment and security), and its frame number is placed into the appropriate places within the PLL and the PMT.

the page is non-resident. Based on the location information in the PLL, and the segment information present in the SCB, a request is made to read the missing page into an empty page frame. The status of the page in the PLL is changed to "in-transit" and the process waits until the page arrives.

the page is pre-paged. This implies that the page was added to the pre-page list as a candidate for removal. The page is removed from the pre-page list and its status is changed back to "resident." The PMT is updated appropriately.

the page is in transit. This implies that some other process has already requested the page. The process waits until it arrives, the requesting process will perform the mapping and this process will awaken to find the page resident.

the page is out-transit. This implies that the page is currently being written out to secondary storage to free the frame in which it was residing. Nothing can be done about reading the page back in until a stabilized image is present on the secondary storage. Therefore, the process waits until the transfer out of memory is complete. When the process is awakened, it will mark the page

as resident and not modified since the resident version corresponds to the version in secondary storage, and then continue, or it will bring the page back in.

In each case, locks are used to ensure consistent results during concurrent accesses.

5.2 Managing Physical Page Frames

In order to efficiently provide empty page frames to satisfy page faults, it is necessary to keep track of the state and use of each page frame. This is accomplished through the use of the *physical page table* or *PPT*. The PPT is organized as an array with the index of each element corresponding to a physical page frame; PPT entry 5 corresponds to frame 5, and so on.

Each entry in the PPT contains information about the status of the page frame, and links to other page frames in the same state. This is accomplished by putting forward and backward link fields in each PPT entry; each link corresponds to the index of the next (or last) PPT entry in a doubly-linked chain of similar entries. There are four such chains threaded through the PPT: the active frame list, the pre-page frame list, the free frame list, and the locked frame list.

The active frame list contains entries referring to page frames which are currently occupied and in use. As pages are brought in in response to page faults, they are added to the end of this list. Each entry in this field also has a pointer to an SCB describing the page, and an offset field which can be used to locate the page's entry in the PLL associated with the SCB.

The free frame list is simply a list of currently available page frames. Requests for empty page frames are satisfied with the entries in this list. We assume that the list is never empty; if necessary, we will suspend all other processes and run a page reclamation process to keep the number of free pages above a minimum threshold.

The locked frame list contains entries corresponding to pages which cannot be thrown out of memory. This includes pages involving active device I/O, pages containing critical code or data (like the PPT!), and pages which are being kept from paging due to performance considerations.

The pre-page list contains entries which are candidates for removal from memory, thus freeing those frames. The page reclamation process will remove entries from the head of the active frame list and add them to the tail of the pre page list, while at the same time tracing down all PMTs which reference this page and marking it as nonresident. Pages are removed from the head of this list and added to the free list after their contents have been written out to secondary storage, if necessary. Pages referenced before they reach the head of the pre-page list get reactivated and moved back onto the active frame list. This whole mechanism implements a form of FINUFO (First In, Not Used, First Out) paging algorithm (Note: if the Vax hardware supported a "referenced" bit in its page tables, this could be avoided!)

6. Secondary Storage, Partitions and the Segment System

6.1 General Structure and Terms

In order to present a consistent, uniform means for the kernel to reference items requiring storage, every item on the system is viewed as having a second type known as *segment*. Segments are basically untyped units of storage which can be read, written, copied, deleted, and moved by kernel code. Paging, whether of a file, a process space, or an object, is always accomplished with the same set

Kernel Structures For Clouds

of segment operations. Copying of items from one place to another is always accomplished with segment operations, and so on.

A segment is *active* if it has been recently accessed or was present in memory before being referenced. The SCB for the segment indicates which processes are referencing it, and which partition driver (see below) needs to be invoked for operations on the segment. These SCBs are allocated and initialized whenever a segment is made active (i.e., referenced and not currently active). The *Active Segment Hash Table* hashes segment capabilities into pointers to the corresponding SCBs.

Each segment representing a permanent item has a version resident on some *partition*. A partition corresponds to some block of space available on a secondary storage device. In general, a partition could exist on a disk, on a tape, or in a block of special memory. Usually, a partition will be blocks of space on disk device(s).

A segment on a partition is composed of a *segment header* and a data area. The segment header contains information about the segment, such as the defined type of the segment (e.g., *client object* or *file*), a capability to the type instance for this segment, time of creation, and other such information. The data space is the actual contents of the segment.

Partitions are composed of a partition header which describes the partition (record size, extent, etc.), a free list whose format is partition-dependent, a directory, directory entries, and data records containing the segments. This structure is more fully described below.

Segments are *recoverable*, *non-recoverable*, or *temporary (volatile)*. Recoverable segments represent items which may be accessed only by actions, while non-recoverable and temporary segments may be accessed by actions or processes. Operations on objects of type *segment* are generally not available to application processes but are always available to kernel code. Any process with the correct capabilities may read any object, whether it is represented as a non-recoverable or recoverable segment. Temporary segments "disappear" on machine crashes and are used for paging space and volatile data structures.

Because of the constraints necessary to ensure the recoverability of objects and to support atomic actions accessing those objects, partitions are not allowed to cross device boundaries. That is, each partition must be fully contained within one device. It is possible to locate the directory and free list for a partition on a device different from the directory entries and data, but this adds a great deal of complexity and delay in the operation of the partition.

Volatile and recoverable segments can be mixed freely within a partition, but some partitions will not be capable of supporting recoverable segments and will be so marked in the partition header. For instance, partitions on a tape are not able to support recoverable objects. In general, a partition supporting recoverable entries must reside on a device which:

- 1) allows random reads and writes.
- 2) does not perform internal buffering of writes except by application choice;
- 3) provides atomic single-record writes.

Condition 3, above, may be relaxed by use of replicated writes to other devices; duplication of writes to implement stable storage is a standard method when dealing with potentially unstable devices and critical applications [Lamp81]

Partitions are added to the system by *mounting*. The operation to mount a partition involves mapping a partition driver to a device driver and scheduler (which may already be active with another partition on the same device). This relationship is shown in figure 6.1. The device driver

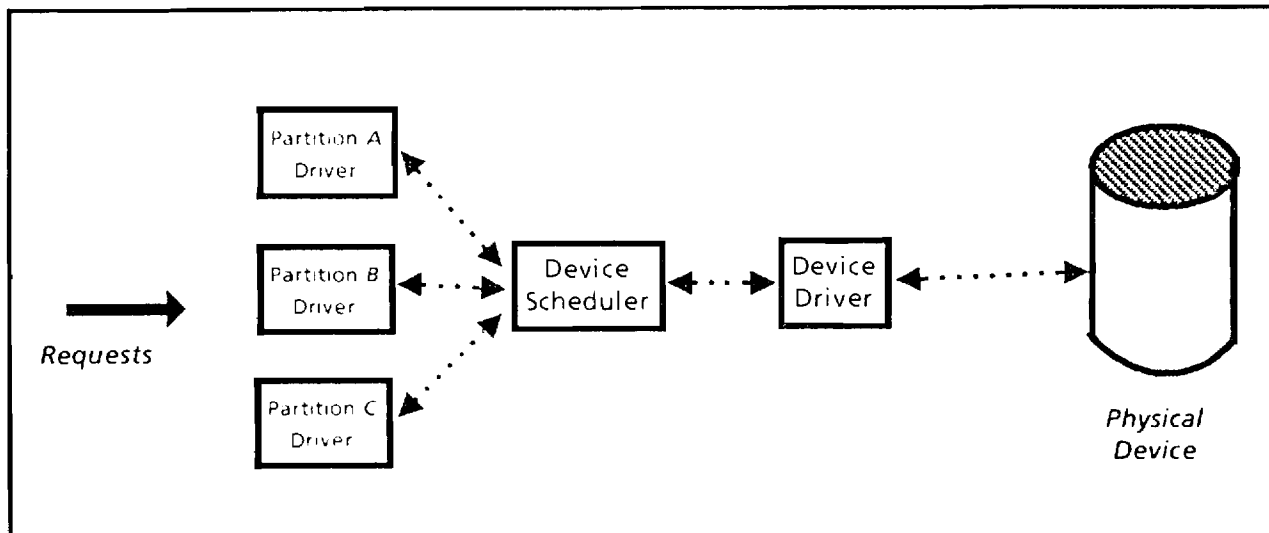


Figure 6.1 -- A mounted partition

module is actually a process which performs operations on the device according to requests provided by the device scheduler module. The driver code is written to take full advantage of individual device characteristics and requirements. This may include implementation of scatter/gather operations, automatic retries of erroneous reads, and so on.

The device scheduler module is designed to take requests from partition drivers and then provide them to the device driver in some orderly, efficient manner. The scheduler module may also contain space to provide buffered reads and writes, and manage these buffers.

Each partition driver accepts requests to read or write a record of a segment in the partition. These requests are then mapped into the appropriate operations on the free list and directory structure, and reformatted into request packets to the device scheduler. The partition driver maps the segment capability into an absolute address in the device based on the segment offset, directory entry, and partition header. Requests to partition drivers may also get modified due to shadowing and recovery considerations. Some requests such as commits and deletes may require that the free list and directory be read or written as well.

Every partition supports operations to *create* a segment, *delete* a segment, *alter* a segment (change its type or maximum allowable size), *read* a page from a segment, *write* a page to a segment, and *truncate* (shorten the working size) of the segment. Each segment is created with a maximum allowable size beyond which it is not allowed to grow. Operations to create a segment or alter a segment require an appropriate capability to the partition as well as the segment involved.

6.2 Recoverable Segments

In addition to objects, partitions need to support recoverability for directory structures and the free lists used in the management of the partition itself. The Clouds storage management organization achieves this goal of recoverability by embedding portions of the free list and directory management in the management of the recoverable segments. This system also allows the shadowing and paging of recoverable objects.

Consider the partition logical organization as shown in figure 6.2. The partition header is always

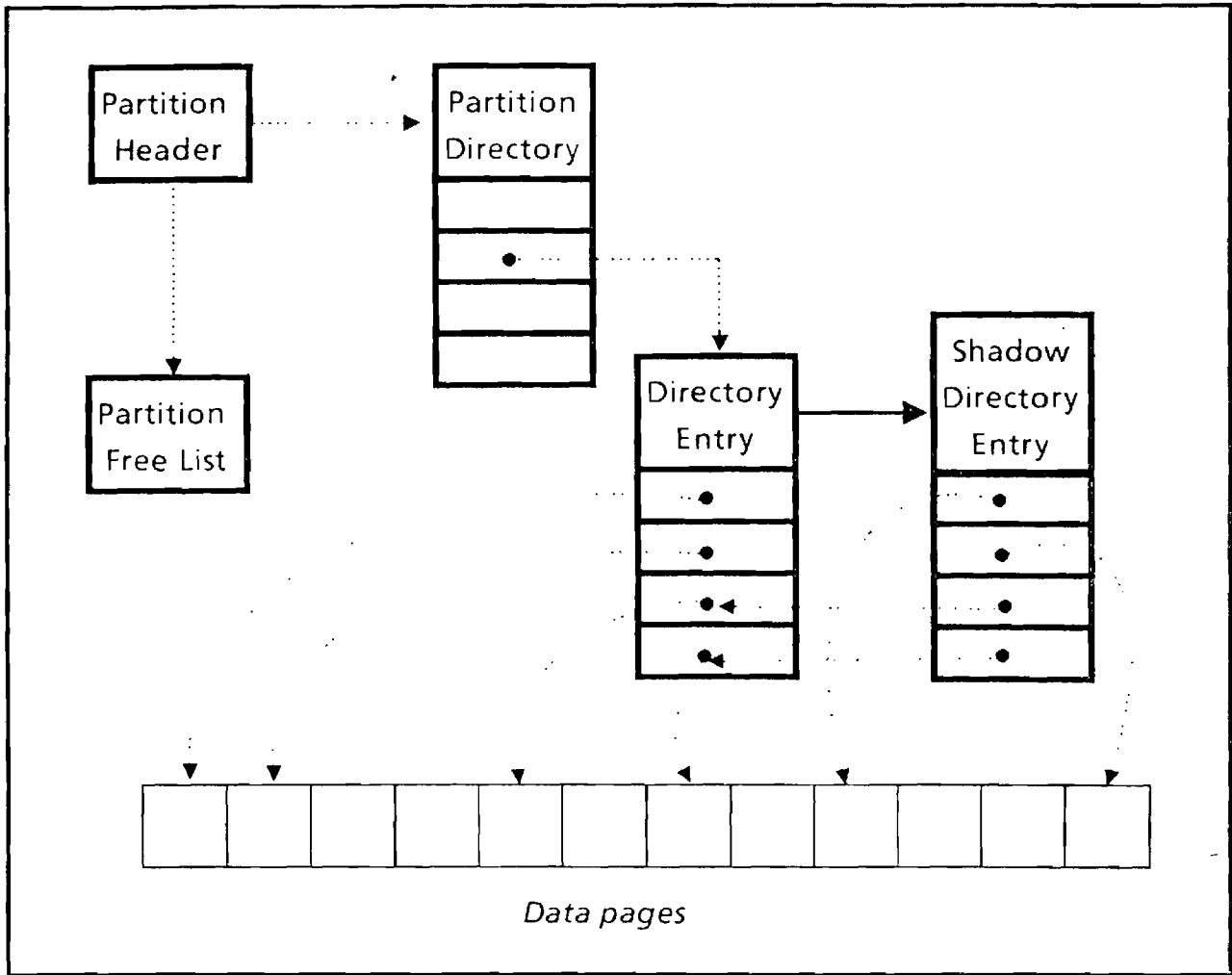


Figure 6.2 -- Partition Organization

found in record zero of the partition. The header contains information about where to find the free list and directory for the partition, and whether the partition supports recoverable segments. The exact form of the free list is not really important as long as it is possible to map it into memory; a bit map is probably the most reasonable form for a disk partition.

The partition directory consists of segment names (sysnames) and the record number within the partition which contains the directory entry for that segment. The partition directory may be many records long, and could be organized as some form of sorted tree rather than as a simple list. To simplify the discussion, we assume here that the directory is simply a linear list of name/address pairs.

Each directory entry consists of name and type information, and a list of the records which make up the segment. This list could also be many records long. Each entry in the list consists of a flag field and a record address which points to the corresponding data record within the segment. In addition to this information, the header contains a field which could contain a pointer to a shadow version of the segment.

When the partition is mounted, the code for the partition driver checks the directory entries to discover uncommitted and precommitted entries (as described shortly). If any such entries are found, their processing is completed. Then a volatile, in-memory copy of the free list is made. The partition directory may also be processed to provide a list of names of objects present on this partition.

When a read reference is made to a recoverable segment in this partition, it is only necessary to use the directory entry to locate the appropriate data page and read it into memory. If the reference is a write reference, then it is necessary to shadow the segment until the referencing action either commits or aborts. Note that in the following description the operations undertaken are all done internal to the partition driver and the calling action never sees anything other than a consistent view of the object.

When the first write to a recoverable segment is done by an action, the partition driver makes a copy of the directory entry on the partition. Each data record pointer in this shadow entry is initially the same as the corresponding pointer in the permanent version. Subsequent writes will be done to newly allocated records containing copies of the corresponding permanent data record. The shadow directory entry is changed to point to these new shadow data pages. After the shadow is created, the permanent version directory entry is changed to include a pointer to the shadow and a status flag indicating that it is being shadowed. If a crash occurs anytime between this point and the time the shadow is committed, the code which checks the directory during partition mount will discover the shadow and remove the pointer.

The records allocated for the shadow segment are all taken from the volatile free list held in memory. The actual free list on the partition is not updated except during the actual commit of the shadow, so any failure simply loses the volatile free list and the records occupied by the shadow remain marked as free in the permanent free list on the partition.

If an abort occurs it is necessary to remove the pointer in the directory entry for the permanent version of the object and mark all of the records of the shadow as "free" in the volatile free list. When a precommit occurs, the status flag in the permanent version is updated from "being shadowed" to "precommitted shadow present". If a crash occurs before a commit or abort, the code which mounts the partition will discover this precommitted segment and then determine whether to complete the commit or abort based on information from other machines in the net.

When a commit occurs, the status of the shadow is changed to "permanent" and its shadow pointer is set to point to the old permanent version (this can be done in a single record write). Next, the partition directory is written to point to the new permanent version. Next, the permanent free list is updated to indicate that the records used in the new permanent version are unavailable and that the records used by the old permanent version are free; the volatile version in memory is likewise updated. Lastly, the pointer in the directory entry for the new permanent version is set to null. If a crash occurs at any point in this processing the commit can be continued from the point of failure when the partition is next mounted.

This scheme assumes that single record writes to the partition are atomic (i.e., uncorruptable during a crash). It also assumes that crashes occur infrequently and that some extra processing when a partition is mounted will not be a difficulty. Using this scheme, aborts and precommits require a single record write. Creation of the shadow requires two record writes in addition to the writes to the data portion of the segment. Commits require approximately three record writes beyond those normally needed to update the free list. Some of these writes may be buffered without affecting the resulting consistency. Thus, this provides an efficient means of supporting recoverable segments and partition structures. Similar algorithms exist for creating and deleting recoverable segments, and for operating on non-recoverable partitions. All of these algorithms are discussed in more depth in [Spaf84] and [Spaf85].

7. Network Communications and RPC

7.1 Network communications

Each machine in the Clouds multicomputer may be connected to one or more other machines in the system by any number and kind of communication channel. The nature of the connections is also immaterial, although it should be obvious that a minimum number of connections of sufficient bandwidth will enhance ultimate performance. The prototype system will have the individual processors connected by a common high-speed backbone and by an Ethernet. Multiple Ethernets and asynchronous communications lines may also be accommodated.

Communication between individual subkernels is handled by a replicated *communications manager*, one per subkernel. Each subkernel makes communications requests through the single interface presented by its copy of the communications manager. Based on the destination and nature of the request, the communications manager chooses the communications channel(s) over which to send the message. The communications manager determines a transmission channel based on message size, message priority, current communications configuration and error counts, and load information. A fixed-size header is prepended to the data portion of the message, and the communication is transmitted over the appropriate medium.

Note that all applications, including the RPC mechanism, define their own protocol for message traffic. The communications manager provides facilities for acknowledging receipt of messages, and for receiving (possibly prioritized) replies to messages. Other than the fixed header block prepended to each message, the communications manager makes no interpretation of the data portion of any message. This allows the operating system and applications software to make direct use of the communications system in whatever manner best suits them.

There is no guarantee that messages are delivered by this communications system to the appropriate software on a remote node. Acknowledgements must be done at a higher level, if desired; the communications driver only supports hardware-level acknowledgements. There are also no guarantees of delivery order or assurances against duplication during transmission. The only assumptions being made in the design of the Clouds communications system are:

- that it is extremely unlikely that the network will be partitioned into isolated segments;
- that if a message arrives at a machine, it is possible to determine if it arrived uncorrupted;
- messages arrive at their destination in small, finite time or else are lost forever;
- the overall probability of corrupted or lost messages is small.

Incoming messages are delivered to the communications manager by individual device drivers and signalled via device interrupts. The communications manager determines the nature of the incoming message and passes it to the appropriate object or process within the subkernel for further action. The communications manager does no protocol checking whatsoever other than determining that the message arrived intact (usually indicated by a hardware status code) and that the message was actually destined for this subkernel. It may use the header information present in each message to update its own internal tables.

The communications manager is made aware of communications channels via an *activate* operation similar to a partition *mount* operation. The communications manager is invoked with a capability to the physical device driver for the channel, and a set of parameters which describe the speed and possible connectivity of that channel. This information is used to help the manager determine when to use that channel for communications. As messages go out and come in over that channel, that information is updated within the internal tables of the communications manager. The first message sent out over an activated channel is a "I'm here, who's there?" message. This is to inform other

systems of the availability of communications over this path, and the responses elicited are used to identify potential destinations for future messages.

7.2 Remote Procedure Calls

One of the operations available on the communications manager is that of the remote procedure call on an object. Normally, this operation is invoked by the object manager when an invocation is attempted on an object which is not present on this machine. The object manager therefore reformats the attempt into an RPC operation on the communications manager. A capability to the object, an operation number and a pointer to a parameter block (usually just the current stack frame of the caller) are the arguments to the call on the communications manager. The communications manager constructs a message consisting of the capability and operation number, and a copy of the parameters to the object (remember that all parameters in object invocations are passed by value). This message is then sent out over the appropriate communications channel(s) and the calling process is blocked until a reply is received. A unique identifier is associated with the message to identify the response.

When the RPC message is received at the site where the object currently resides, the communications manager requests a *cohort* process from the process manager. The cohort is dispatched with a copy of the object capability and operation number, and with a pointer to the copy of the parameter block which comprised the remainder of the message. If the request was on behalf of an action, the cohort assumes the action identity of the calling action via a call to the object manager. Next, the cohort copies the parameter block to its stack and invokes the object as if the call had originated locally with the cohort. When the invocation returns to the cohort, it calls the communications manager to formulate a reply to the original request. The reply is constructed with the values and error codes returned by the invocation, if any, and with the unique identifier provided with the incoming RPC message. This reply message is then sent out and the cohort is reclaimed by the process manager.

When the reply message is received at the original machine, the communications manager alters the state of the blocked requesting process to indicate the location of the reply message, and then the process is unblocked. It retrieves the reply information, alters its stack and registers as indicated by the reply, and then returns. The whole process of call and return looks exactly like a local (although possibly slow) invocation.

8. Object Searching and Invocation

One of the most important features of objects in Clouds is that every invocation embodies an implicit search. No assumptions are made about the location of objects. In fact, it is entirely possible that objects may move from machine to machine between invocations. The implicit search also allows more dynamic use of cloned objects and alternate communications pathways.

When an object is invoked, the object manager searches the Active Object Table for an entry matching the name of the object. If such an entry is not found, then a new entry is allocated and added with a set of default values. If the name is found in the table, then the entry contains a pointer to one of two things: an OCB (Object Control Block) or a search module. If the pointer is to an OCB, then the object is present on this machine, and is either currently active or was just recently active. In either case, the OCB contains all of the necessary information to bring the object into memory and map it into the virtual address space of the requesting process.

If the entry for the object is a pointer to the default (initial) search module, then all of the local partitions are searched to see if the object is present locally. If it is found on a local partition, then an OCB is allocated and initialized and the invocation proceeds as above. If the object is not found

Kernel Structures For Clouds

locally, then the pointer is altered to point to the network search module and that module is then activated.

The network search module uses available information about the configuration of the network and the available communication paths to seek the object on some set of remote machines. This search operation may actually be done as a "search and perform" operation to save time, with the parameters for the RPC included with the search message. If a remote site receives such a search message, it attempts to find the object locally. If the object is found, it is made active locally and the operation is performed as a normal RPC.

If the network search module fails to locate the object on any remote machine it returns a simple "not found" indicator to the requesting process, aborting it if it is an action. It is not possible to determine at this point whether the object does not exist, exists on a processor which is currently not communicating with this machine, or whether the object is currently not available due to action visibility constraints.

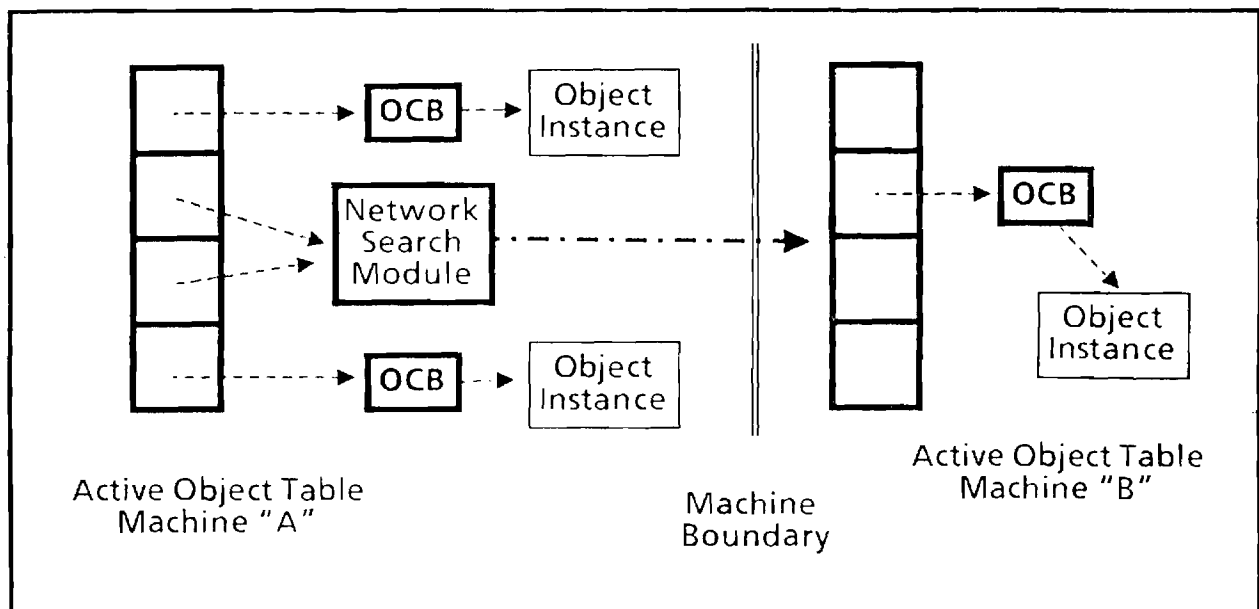


Figure 8.1 -- Locating an object

The search strategy may well be optimized somewhat through the use of hints. Once an object has been found and made active it is possible to include a hint with the entry in the active object table on the calling machine. Future references to the object can be tried first on the hinted-at machine since that is the last known location of the object. It may also be possible to derive hints in other ways; this area will be the subject of further research by the Clouds group. [Pitts85]

9. Kernel Interface

Conceptually, the kernel interface extends across machine boundaries and exists on each processor within the Clouds system. In actuality, the interface is replicated on each machine. In our prototype, the interface consists of service calls which change the processor mode to the kernel state and then examine the arguments for validity. Kernel calls (including object invocation) use a protected per-

user stack to hold return state information. Calls by code within the kernel to other parts of the kernel are done directly and avoid the overhead of a system trap.

In general, each kernel operation requires one or more capability parameters specifying what is to be done. The kernel interface maps those capabilities to simpler operations to be performed by specific subkernels. Most kernel operations are done by actions dispatched by the interface. This allows better error containment and prevents kernel operations from being only partially completed. The actions so spawned operate independently of the requesting process or action. This mechanism helps simplify the design of the distributed aspects of the kernel, especially when dealing with kernel services being performed on remote systems which may possibly fail in the midst of the operation.

As an example, consider a request to the kernel to move a file from one partition to another. The request to the kernel would include a capability to the file and a capability to the destination partition. The request does not require the specification of any specific machine names or locations. The kernel will locate the file and partition based on the provided capabilities. The movement of the file will occur as part of an action, with the copy being done with segment operations to read and write pages of the file. Should the communications channel or one of the processors fail during the transfer, the action will be aborted and the partially transferred file will be erased from the destination partition. If the destination partition does not have enough space for the file, the action will abort and the space will be freed. Other errors act in a similar manner with appropriate error codes being returned to the caller, if possible.

This method of structuring the interface also allows the system to be expanded to other processors which may employ a different underlying architecture. As long as it is possible to create an action somewhere in the Clouds system then the calls through the kernel interface can be attempted at some location; the operation of the kernel interface is independent of the location of the requester since all location information is contained within the capabilities passed as arguments.

10. Conclusion

This paper has presented an overview of the internal structure of the Clouds kernel. This presentation has also given an indication of how these structures will interact in the prototype Clouds implementation. Many specific details have yet to be determined and await experimentation with an actual working system.

11. References

- [Alle83a] Allechin, J. E., "An Architecture for Reliable Decentralized Systems," Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also released as technical report GIT-ICS-83/23)
- [Alle83b] Allechin, J. E., and M. S. McKendry, "Synchronization and Recovery of Actions," Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983
- [Alme83] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," University of Washington Department of Computer Science, Technical Report 83-10-05, October 1983

Kernel Structures For Clouds

- [Birm84] Birman, K , *et.al.*, "Implementing Fault-Tolerant Distributed Objects," Cornell University Computer Science technical Report 84-594, March 1984
- [Jone79] Jones, A. K., "The Object Model: A Conceptual Tool for Structuring Software," Operating Systems: An Advanced Course, Springer-Verlag, NY, 1979, pp. 7-16
- [Lamp81] Lampson, B. W., "Atomic Transactions," Distributed Systems: Architecture and Implementation, Springer-Verlag, NY, 1981, pp. 246-265
- [Lisk83] Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS*, Vol. 5, No. 3, July 1983
- [McKe83] McKendry, M. S., J. E. Allehin, and W. C. Thibault, "Architecture for a Global Operating System," IEEE Infocom, April 1983
- [McKe84] McKendry, M. S., "Clouds: A Fault-Tolerant Distributed Operating System," IEEE Distributed Processing Technical Committee Newsletter, 1984
- [Pitt85] Pitts, D. V., "Naming and Searching in a Distributed Operating System," PhD Thesis, School of Information and Computer Science, Georgia Institute of Technology, *in progress -- 1985*
- [Rash81] Rashid, R. F., and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," Proceedings of the 8th Symposium on Operating Systems Principles (ACM SIGOPS), December 1981
- [Spaf84] Spafford, E. H., "A Consistent File System for Clouds," Technical Report, School of Information and Computer Science, Georgia Institute of Technology, *in progress -- 1984*
- [Spaf85] Spafford, E. H., "Kernel Structures for a Distributed Operating System," PhD Thesis, School of Information and Computer Science, Georgia Institute of Technology, *in progress -- 1985*
- [Weih83] Weihl, W. and B. Liskov, "Specification and Implementation of Reilient, Atomic Data Types," Symposium on Programming Language Issues in Software Systems, June 1983
- [Wulf74] Wulf, W. , *et al.*, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, Vol 17, No. 6, June 1974

**A Support Architecture for
Reliable Distributed Computing Systems**

*Final Project Report
May 15th 1990*

From:

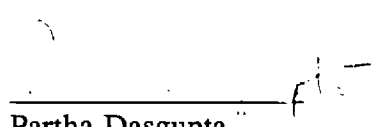
Georgia Tech Research Corporation
Atlanta, Georgia 30332

To:

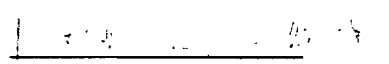
National Aeronautics and Space Administration
Langley Research Center

Grant No.: NAG-1-430

Endorsements:



Partha Dasgupta
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2572



Richard J. LeBlanc Jr.
Co-Principal Investigator
School of ICS
Georgia Tech, Atlanta GA 30332
(404) 894-2592

A Support Architecture for Reliable Distributed Systems

1. Introduction

The Clouds project at Georgia Tech has been funded by NASA since 1981. During this period we have designed and implemented a distributed object-based operating system, which has a novel structure and supports a unconventional systems-programming style.

The NASA funds received this far had a major impact on the formation of the Clouds project and in the area of research in distributed operating systems. Using these funds we have achieved recognition in the research community, published several papers and many technical reports, implemented the first version of the Clouds system, and most notably, were selected by the National Science Foundation as one of the recipients of the 1987 CER (Coordinated Experimental Research) grants.

The first Clouds kernels was a monolithic, native kernel that ran on VAX-750 computers. This version was implemented in the 1983-86 timeframe and was used till mid 1987. After this kernel served its purpose, we redesigned the implementation and have built a new minimal kernel. The redesign effort was started in late 1987. Since 1988 we have redesigned Clouds, implemented the system as a minimal kernel, added system services and user service and have built a usable distributed system. The current version runs on Sun 3/60 machines.

This report provides a brief overview of the Clouds implementation and the project status.

2. The Clouds Project

The Clouds project is a distributed operating systems project. The goals of the project are:

- The operating system will be distributed over several sites. The sites will have a fair degree of autonomy. Yet the distributed system should work as an integrated system. Thus the system should support location independency for data, users and processes.
- To keep the system simple, the user environment must be buit around a straightforward, simple and elegant paradigm. The paradigm must encapsulate storage, processing and distribution.
- The users should be shielded from both the configuration of the system (site independence) as well as its failure modes. For example, if the site a user is connected to fails, he should be transferred to an active site transparently.
- Reliability is a key requirement. Large distributed systems use significant number of hardware components and communication interfaces, all of which are prone to failures. The system should be able to function normally even with several failed components. Data should be kept consistent at all times.

- Distributed systems often have dynamic configurations. That is, newer hardware gets added, or faulty hardware is removed. The system function should not be hampered by such maintenance chores. Thus the system should be dynamically reconfigurable.
- Many of the above functions can be implemented on conventional systems, but would make the system extremely slow. Thus efficiency is an important design criteria.

The above goals have resulted in a elegant and feasible design of a object-based distributed operating system.

3. The Design of Clouds

The Clouds design hinges on a simple paradigm of system structure, using two basic primitives namely objects and threads. The following are the salient points of the design.

- An object-based, passive system paradigm is used as the basic architecture. All system functions, data, user programs and resources are encapsulated in passive objects. These objects can be invoked at appropriate entry points by threads.
- Threads and the only active entities in the system. A thread is a path of execution that visits entry points in objects in a procedure-call fashion and may span several objects in several sites. Also several threads may concurrently execute in the same object.
- The objects in Clouds represent nearly everything the system has to offer. The site independence philosophy is implemented by making the object name space (system names) flat and site independent. When a thread on any machine invokes an object located anywhere, no site names are used. Hence the location of any particular object is unknown to a thread.
- Efficiency has been of concern. To this end the Clouds kernel (also called *Ra*) is a small minimal kernel that support threads and object invocation. The objects are handled by the virtual memory system. This design is kept simple without losing generality of flexibility, and is expected to be efficient.

The Clouds paradigm has been augmented to support features for reliable computation. The two major features added to support reliability is the notion of consistency-preserving invocations and replicated computations.

- The consistency support system is a user programmable set of tools which are triggered by declaring the entry points of the objects with a consistency level. The consistency levels include global, local and standard. These labels give rise to threads (upon invocation) that have specific semantics. A complete outline of the semantics is outside the scope of this report. The allowed semantics provide the support needed for writing robust programs.
- Reliability is further enhanced with replication at the data as well as computation level. This allows the system to recover from a certain fixed number of faults with minimal overhead.

4. The implementation of Clouds

The implementation of Clouds is largely complete. The implementation includes a minimal kernel called Ra. The Ra kernel provides low level memory management and scheduling support. The rest of the operating system is programmed at above the kernel level using a facility called system objects. The following is a brief outline of the services provided by the system objects:

- (1) *Buffer Manager*: This is a low-level service that handles the buffer allocation, deallocation, packaging and unpacking needs of a variety of system service routines.
- (2) *Object Manager*: The Ra kernel provides the mechanisms necessary for handling objects in the operating system. The Object manager handles most of the policy including construction, activation and invocation of objects.
- (3) *Network Driver* The basic service used by all distributed facilities of the operating system is the Ethernet driver that allows packet communication over a local area network.
- (4) *The Transport Protocol (RaTP)* RaTP is a reliable message transport protocol that is used by all communication activities. RaTP is a connectionless, lightweight protocol that can handle message transaction oriented (synchronous) communication. This is used to support a variety of functions detailed below.
- (5) *Thread Manager* A Clouds thread is a distributed computation. The thread manager handles the remote processing abilities of threads.
- (6) *RPC Handler* The Clouds remote procedure call facility is built on top of RaTP in conjunction with the thread manager. Allows remote objects to be executed.
- (7) *Network Disk Manager*: The Clouds machines are equipped to run diskless. The disk services are provided from file-servers. Thus the network disk drivers create virtual disks for object storage and swapping for Clouds machines. The Network disk drivers run on top of RaTP.
- (8) *Swap Manager* The local swapping facilities of each Clouds machine is handled through the swap manager. The swap manager implements the paging facilities and ties them to a swap partition on a remote disk via the network disk manager.
- (9) *DSM Partition*: Clouds supports a Distributed Shared Memory facility. This makes all objects in the system appear local to all machines running Clouds. The DSM partition provides this virtual object access capability along with coherence support to enforce single-copy semantics.
- (10) *Distributed Locking Handlers*: Objects have local control over synchronization of access to its private data, even when accessed by several threads running on different machines. This is handled by the distributed locking service. The distributed locking service is accessed by programmers as semaphores declared within the object.
- (11) *User I/O Handler*: User I/O is handled by Clouds objects as ASCII I/O to user terminals. User terminals are in fact windows running under SunWindows on Unix machines. The user can run Clouds applications via a Unix window. All I/O

is redirected to the appropriate window over the network. The User I/O manager supports printf's and scanf's from objects to the user pseudo-tty.

In addition to the above services, Clouds provides a modified C++ programming environment called CC++. We support a CC++ compiler and mechanisms of generating objects and instances.

5. Progress Report

The Clouds research team currently consists of five academic faculty, two research scientists and about 8 graduate students. Most of the funding is from the National Science foundation. The NASA support is used to augment the NSF support and currently it supports one Ph.D. student and some faculty time.

The above system functions have been implemented and tested. We are currently augmenting the implemented functions with support for programming environments and consistency mechanisms. Design work in the areas of naming, location, object-oriented programming support and distributed management is in progress.

6. Bibliography

The following publications include contributions by persons supported by by NASA:

- *The Structure of the Clouds Operating System*. P. Dasgupta and R. J. LeBlanc. 1989 Workshop on Operating Systems for Mission Critical Computing, Sept 19-21, College Park MD. (To appear as a chapter in *Operating Systems for Mission Critical Computing*, ACM Press.)
- *The Five Color Protocol: Non Two Phase Locking in Databases*. P. Dasgupta and Z. M. Kedem. ACM Transactions of Database Systems June 1990.
- *The Clouds Distributed Operating Systems*. Joint ORSA/TIMS Meeting, P. Dasgupta. New York, October 31st 1989. (Invited Paper).
- *Linking Consistency with Object/Thread Semantics: An Approach to Robust Computations*. R. Chen and P. Dasgupta. 9th International Conference on Distributed Computing Systems, Newport Beach CA, June 5th-7th, 1989.
- *The Clouds Distributed Operating System*. P. Dasgupta, R. J. LeBlanc and W. F. Appelbe. Proceedings of the 8th International Conf. on Distributed Computing Systems. June 14th-16th 1988.
- *Fault Tolerant Computing in an Object Based Distributed Operating System*. M. Ahamad, P. Dasgupta R. J. LeBlanc and C. T. Wilkes. Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems, Williamsburg, March 18-19, 1987.
- *Structuring a Distributed Operating System: The Clouds Approach and Experience*. P. Dasgupta and R. J. LeBlanc. IEEE Computer Society Workshop on Design Principles for Experimental Distributed Systems, Purdue University, October 16th-17th, 1986.

- *A Probe-Based Fault Tolerant Scheme for an Object Based Operating System.* P. Dasgupta. Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications. Portland OR, Sept. 1986.

Appendix

Selected Papers



The Structure of the Clouds Distributed Operating System.*

Partha Dasgupta and Richard J. LeBlanc Jr.

School of Information and Computer Science
Georgia Tech
Atlanta, GA 30332

Abstract

A novel system architecture, based on the object model, is the central structuring concept used in the Clouds distributed operating system. This architecture makes Clouds attractive over a wide class of machines and environments. Clouds is a *native* operating system, designed and implemented at Georgia Tech, and runs on a set of general purpose computers connected via a local area network.

The system architecture of Clouds is composed of a systemwide global set of persistent (long-lived) virtual address spaces, called *objects* that contain persistent data and code. The object concept is implemented at the operating system level, thus presenting a single level storage view to the user. Lightweight threads carry computational activity through the code stored in the objects.

The persistent objects and threads gives rise to a programming environment composed of shared permanent memory, dispensing with the need for hardware-derived concepts such as file systems and message systems. Though the hardware may be dis-

tributed, and may have disks and networks, the Clouds provides the applications with a logically centralized system, based on a shared, structured, single level store.

The current design of Clouds uses a *minimalist philosophy* with respect to both the kernel and the operating system. That is, the kernel and the operating system support a bare minimum of functionality. Clouds also adheres to the concept of separation of policy and mechanism. Most low-level operating system services are implemented above the kernel and most high level services are implemented at the user level. From the measured performance of using the kernel mechanisms, we are able to demonstrate that efficient implementations are feasible for the object model on commercially available hardware.

Clouds provides a rich environment for conducting research in distributed systems. Some of the topics addressed in this paper include distributed programming environments, consistency of persistent data and fault-tolerance.

1. Introduction

The *Clouds* project is an ongoing distributed operating system project at Georgia Tech. The *Clouds* operating system was first designed in 1983 [Al83, Mc84]. The implementation was started in 1984 and was completed in 1986 producing the first version of the software (*Clouds v.1*) [Sp86, Pi86,

Ke86]. In mid-1987 we started designing the second version of *Clouds* and the kernel for this version has been completed in mid-1988 [DaLeAp88, BeHuKh88, BeHuKh89]. This version is called *Clouds v.2*. In this paper we discuss the basic philosophies and techniques that have been developed through the experience gained in the *Clouds* project as well as the implementation and reasearch issues

* This research was partially supported by NASA under contract number NAG-1-430 and by NSF grants DCS-8316590 and CCR-8619886 (CERII program).

What can objects do?

Objects can be used for most everything. An object is an encapsulation that includes not only long-lived data; but a set of operations (invocations) on the data, or an environment under control of the object.

In the simplest form, objects can be considered modules that provide a function or service. For example a shared library, a data server, a program fragment and so on.

Objects can also provide specialized services. For example, a sensing device can be represented as an object and an invocation can be used to gather data from the device, without having to know about the mechanisms involved in accessing the device, or even the location of the device. Similarly, terminal I/O

is effectively handled by a terminal object (with read and write operations defined on it).

Objects can be active. If a process is started in an infinite loop in an object, this process can monitor the environment of the object and inform some other entity (another object) in case of need. This is particularly useful in conjunction with an object managing sensory monitoring devices.

Objects are a simple concept with a major impact. From general purpose programming needs to quite specialized applications, the object can be used for almost every need, and yet provide the same, simple procedural interface to the rest of the system.

system. In addition it supports an object-based system structuring paradigm.

The paradigm used for defining and implementing the system structure of the *Clouds* system is an object/thread model (section 2). This model provides threads to support computation and objects to support an abstraction of storage. This model has been augmented to support atomicity of computation and to provide support for reliable programs [Al83, ChDa89, AhDaLe88].

1.3. Project Overview

The first version of the *Clouds* operating system has been implemented and is operational. This version is referred to *Clouds v.1*. This was used as an experimental testbed by the implementors. This implementation was successful in depicting the feasibility of a native object-based operating system, supporting the *Clouds* paradigm. The implementation of *Clouds v.1*, in retrospect had some flaws that resulted in poor performance and non-portability of the kernel. Experience with *Clouds*

v.1 taught us that the approach works; it also taught us how to do it better.

The lessons learned from the previous implementation have been used to redesign the kernel and build a new version called *Clouds v.2*. The basic system structuring paradigm used in *v.1* and *v.2* remain the same. Most of the design objectives mentioned earlier are the same as before. However some of the goals and most of the design and implementation of the system has changed. *Clouds v.1* was targeted to be a testbed for distributed operating system research. *Clouds v.2* is targeted to be a distributed computing platform for research in a wide variety of areas in Computer Science system research.

The structure of *Clouds v.2* consists of a minimal kernel called "*Ra*"*, and a set of system-level objects providing the operating system services. *Ra* supports a set of basic function of the system: virtual memory management, system object support and low-level scheduling. The system objects provide other systems services (user object management, synchronization, naming, atomicity

* After the Egyptian Sun-God.

Page missing from report

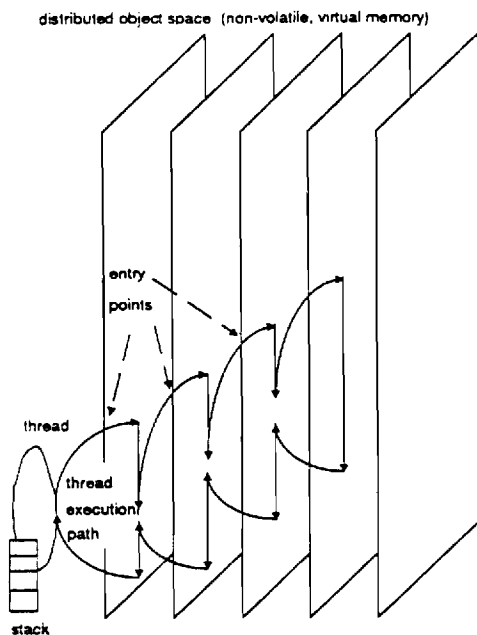


Figure 2: Distributed Object Memory.

segment is accessible by the code in the code segment, but not by any other object. Thus the object has a wall around it which has some well-defined gateways through which activity can come in. Data cannot be transmitted in or out of the object freely, but can be moved as parameters to the code segment entry points (see the discussion on threads). A *Clouds* object is shown in Figure 1.

Clouds objects can be defined by the user or defined by the system. Most objects are user-defined. Some examples of system-defined objects are device drivers, name-service handlers, communication systems, systems software, utilities, and so on. The basic kernel is not an object; it is an entity that provides the mechanism for the support of object invocations. A complete *Clouds* object can contain user-defined code and data, system-defined code and data that handle synchronization and recovery, a volatile heap for temporary memory allocation, a permanent heap for allocating memory that will remain permanent as a part of the data structures in the object, locks, and capabilities to other objects.

Though *Clouds* objects can be created, deleted and manipulated individually, the operating system is designed to support a class and instantiation mechanism. This mechanism allow users to derive object-classes from a set of parent classes, as well as create object-instances of an object-class. Further details are dependent on the programming language and environment used, and are beyond the scope of this paper.

2.2. Threads

The only form of activity in the *Clouds* system is the thread. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. Threads can span objects, and can span machine boundaries. In fact, machine boundaries are invisible to the thread (and hence to the user). Threads are implemented in the *Clouds* system as lightweight processes that have a stack space but no data space. A thread that spans machine boundaries is implemented by several processes, one per site.

Upon creation, a thread starts up at an entry point of an object. As the thread executes, it executes code inside an object and manipulates the data inside this object. The code in the object can contain a call to an operation of another object. When a thread executes this call, it temporarily leaves the calling object, enters the called object, and commences execution there. The thread returns to the calling object after the execution in the called object terminates. The calls to object entry points are called *object invocations*. Object invocations can be nested. The code that is accessible through an entry point is known as an *operation* of the object.

A thread executes by invoking operations defined inside many objects. Unlike processes in conventional operating systems, a thread often cross boundaries of virtual address spaces. Visibility within an address space is, however, limited to that address space, thus the thread cannot address any data outside its current address space. Control transfer between address spaces occurs though object invocation and data transfer between address spaces occurs through parameters to object invocation. A thread in

Page missing from report

Second, the storage mechanism used in this object-based environment is quite different from that used in the conventional operating systems. Conventionally, the file is the storage medium of choice for data that has to persist, especially since memory is tied to processes and processes can die and lose all the contents of their memory. However, memory is easier to manage, more suited for structuring data and essential for processing. The object concept merges these two views of storage, to create the concept of a set of shared, structured, permanent virtual spaces.

Just as *Clouds* does not have files, it does not provide user-level support for disk I/O. In fact there is no concept of "disks" or such I/O devices (except user terminals). The system creates the illusion of a large virtual memory space that is permanent (non-volatile), and thus the need for using peripheral storage from a programmer's point of view, is eliminated.

Many distributed systems are message-based, and hence use messages as the paradigm of choice. In the object-thread paradigm, like the need for I/O, the need for messages is eliminated. Threads need not communicate through messages. Messages can be simulated using objects implementing bounded buffers for applications written using a message-oriented algorithm. Thus ports are not supported. This allows a simplified system management strategy as the system does not have to maintain linkage information between threads and ports.

Objects provide an easy to use abstraction for shared memory, a special case of which ("problem-oriented shared memory") is recommended by Cheriton as a powerful tool for programming distributed systems [Ch86]. Shared memory (consistent or not) is seen by many as a better concept than messages for programming distributed systems, e.g. Linda [Ge85].

To summarize:

- The *Clouds* system is composed of named address spaces (objects).

- Activity is provided by threads moving amongst the population of objects through invocation.
- Data flow is implemented by parameter passing.

The system thus looks like a set of *permanent address spaces* which support control flow through them, constituting what we term *object memory*.

The basic paradigm discussed above is the common link between *Clouds v.1.* and *Clouds v.2.* Both are identical in this regard. Most of the other features of the two are different.

3. *Clouds v.2.*

The development of the second version of *Clouds* was undertaken for several reasons:

- The implementation of *Clouds* on the VAX machines suffers from inherent problems due to the VAX architecture. Other machines are better suited for implementing an object system due their virtual memory mechanisms being more compatible with the demands of a persistent object system [RaKh88b].
- The *Clouds v.1.* kernel was not written to be portable. Since the VAX did not prove to be an effective target machine for *Clouds*, porting entailed major rework.
- The kernel was monolithic, large and hard to modify. This attribute encouraged us to think about the minimal kernel approach.
- We learned how to manage virtual memory better, and could identify a lot of situations where the need for separation of policy and mechanism was necessary for further research into both policy and mechanisms.

The redesign of *Clouds* not only prompted a redesign of the kernel, but a rethinking of all of the system functions. The only aspect of the system that remained unchanged in the redesign phase was the basic object/thread paradigm described in sections 2.

Page missing from report

tual machine is shown in Figure 3. The abstractions supported by the *Ra* virtual machine are:

- **Segments:**

A segment is a contiguous block of virtual memory and its contents. The contents of a segment are an uninterpreted byte-sequence. Segments are explicitly created and persist until destroyed. They have systemwide unique sysnames and they have storage attributes, such as zero-filled, volatile, persistent, recoverable etc.

- **Windows:**

A window defines a contiguous range of addresses in a virtual space. This range is associated with a range of contiguous addresses in a segment. Windows are used to create virtual spaces out of segments. Windows have protection attributes such as read-only, read-write and so on.

- **Virtual Spaces:**

A virtual space is the abstraction of a complete address space. A virtual space consists of a set of windows into one or more segments. Thus a virtual space is a set of subranges of segments, coalesced into one address space. The virtual space is the representation of an object, when the object is being actively used. The composition of a virtual space is shown in Figure 4.

- **Partitions:**

Partitions are containers for segments and provide non-volatile data storage for the segments. A segment lives in exactly one partition. When a segment is in use, it can be demand paged, in and out, of its partition. Partitions can exist on any machine in the network, including machines not running *Ra*. (e.g. *Ra* segments could reside on a Unix machine running a *Ra* partition server.) Though partitions are a part of the *Ra* virtual machine, partitions are imple-

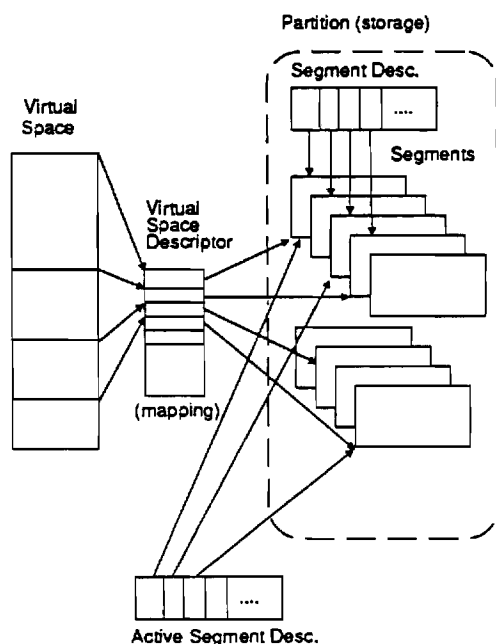


Figure 3: The Structure of *Ra*.

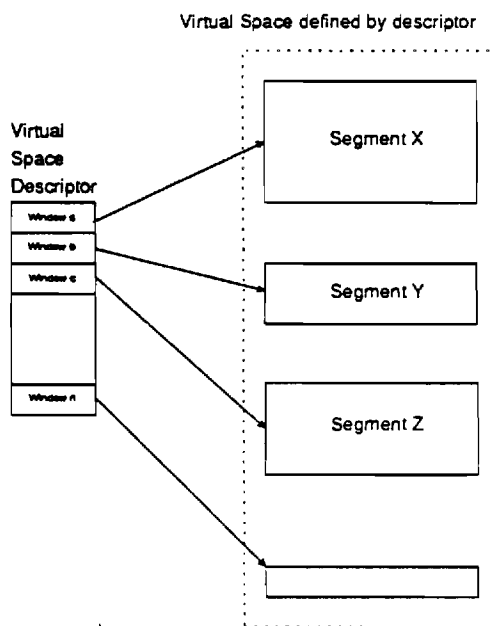


Figure 4: *Ra* Segments and Windows.

Page missing from report

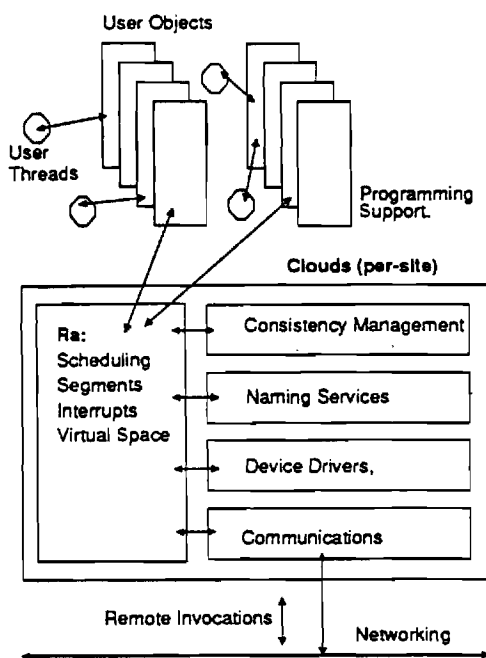


Figure 7: The Clouds/Ra Environment.

on top of the minimal *Ra* kernel. A system object has the looks like a user-level *Clouds* object, but exists in the system (protected) space of the machine. Operations in system objects can be invoked by user objects, on behalf of user objects by *Ra*, or by system *IsiBas* that run in the system space.

Though most of the system objects are optional, *Ra* requires some *essential system objects*. The essential system objects are a buffer manager, a network driver, a network protocol handler and a partition. A *Ra* system with disks will need a disk driver. A diskless *Ra* system the partition communicates to a segment server over a network, creating the illusion that the segments actually reside on the local site.

If the *Ra* kernel is used to support *Clouds*, it needs at least an object manager. Some of the optional system objects include consistency manager, replication manager, system monitor and so on. All the essential system objects have been implemented and tested, and some of the optional ones are being designed.

The structure of the system memory that contains the kernel, system objects and kernel classes is shown in Figure 5; Figure 6 shows the storage and activity hierarchies in *Clouds*; and the structure of the *Clouds/Ra* system is shown in Figure 7. The design of *Ra* is explained in further detail in [BeHuKh89].

3.1.3. Objects in v.2.

The *Clouds* objects are implemented in v.2. through a system object called the object manager. The object manager manages objects, creates and deletes them and provides the object invocation facility.

An object is a set of windows into a set of segments and is implemented using a *Ra* virtual space. The information regarding the windows and segments is stored in another segment called the object descriptor. The *sysname* of the object descriptor segment, plus requisite access rights is the capability of the object.

Objects that are in *Ra* virtual spaces can be invoked. Information on the most recently used objects is contained in an *activated object table* in the object manager. When a thread invokes an object, the object manager first checks the object table. If the information is present the invocation proceeds. Otherwise the object table information is built by reading in the object descriptor.

3.1.4. Invocation using RPC and DSM

The object naming scheme in v.2. is the same as in v.1. (system-wide capabilities). The remote object location scheme is currently a subject of research. At present we are using the cache/search/invoke scheme used in v.1. but may change to a multicast-based scheme [Pi86, Sp86, AhAm87, Be88].

The object invocation scheme is handled by the object management system (a system object) and not the kernel. If the object is available in a local partition, the object manager maps the virtual space of the called object into the address space of the thread, in the place of the calling object, and starts the thread at the entry point in the called object.

Page missing from report

the DSM partition. The DSM partition implements all the protocols (including coherency protocols) necessary for the Distributed Shared Memory function. The DSM server currently runs on Unix and provides service to *Clouds* machines.

3.2. The Ra Transport Protocol

All communications between machines running *Clouds* as well as the communications between diskless *Clouds* machines and storage servers on Unix are handled through the Ra transport protocol or RaTP. RaTP is a connectionless, reliable protocol that efficiently provides client-server communications using the message transaction model. RaTP is a minimal protocol and provides only the functionality necessary for RPC and DSM based object invocations. More functionality will be layered on top of RaTP if necessary. RaTP is quite similar to (but much simpler than) the VMTP protocol used by the V-System [Ch86].

3.3. Programming Support

Programming objects in *Clouds v.2.* can be done through any structured, procedural language. The linker and the loader have to be modified to generate an object in the proper format, from compiled modules of the language.

However most of the facilities of *Clouds* will be inaccessible to the programmer using an off-the-shelf language. For this reason, we are experimenting with a pre-processed (and post-processed) version of C++. The features necessary for supporting programming of *Clouds* objects are:

- Defining an object class from existing classes, or from the default parent class.
- Declaring the entry points.
- Mechanisms for instantiation of object instances from class instances.
- Labeling the entry points with consistency labels (see section on atomicity.)
- Declaring data segments.

- Support for dynamically changing the size of the data segments.
- Access to the object naming functions, capability storage and manipulation.
- Access to the semaphores and locks provided by *Clouds*.
- Support for creation and deletion of data segments and using data segments to create objects.

In theory, all these can be built into an existing programming language. Some of the features (especially the last three) can be provided by runtime libraries.

The C++ program segments that define a *Clouds* object is compiled on Unix and a set of *Clouds* segments are generated. The segments are loaded on the *Clouds* segment server(s) and can be invoked from *Clouds*.

Another set of programming tools that will be provided by *Clouds* are utility objects and predefined objects. Utilities will be similar to utilities in any operating system, and will be provided by objects that will be a part of the *Clouds* environment. Predefined objects will be object classes which can be instantiated by the user to provide some services that are deemed useful. We are in the preliminary stages of research in this area.

3.4. User Interfaces

We plan to use Unix and X-windows as our interface to *Clouds*. Unix programs can make use of *Clouds* facilities through invocation support provided by a *Clouds* library on Unix. Also, *Clouds* utilities will be available under X-windows. This will have several implications:

Firstly, *Clouds* can be treated as a back-end system to the Unix workstation, for distributed processing, computations, object-oriented programs and atomic programs. All these facilities will be available to Unix programs and the user.

Second, the user can access *Clouds* utilities through the X-window system, and thus

Page missing from report

terleave with other s-threads, as well as with cp-threads.

There are two varieties of cp-threads, namely the *gcp-thread* and the *lcp-thread*. The *gcp-thread* semantics provide global (heavyweight) consistency and the *lcp-thread* semantics provide local (lightweight) consistency.

All threads are s-threads when created. The handling of cp-threads are programmed by the following scheme. All operations defined on objects in *Clouds* are tagged with a consistency label; the labels used are:

- Globally-Consistent (gcp)
- Locally-Consistent (lcp)
- Standard (s)
- Inherited (i)

An object can have any number of different labels on its operations. Also, the same operation may have multiple entry-points, labeled at different atomicity levels. A s-thread executing a gcp or lcp operation converts to a gcp or lcp-thread. A thread entering a lcp entry point commits its updates (inside this object) just before it exits the object. This provides intra-object consistency rather than the inter-object consistency provided by the gcp operations, and thus is a cheap method of updating one object atomically. Locking and recovery are automatic.

The standard entry points do not support any locking or recovery. They can make use of "best-effort" semantics. They can also be used for non-traditional purposes such as peeking at incomplete results of actions (as they are not hindered by locking and visibility rules of actions). Locks are available for synchronizing s-threads, but recovery is not supported.

A thread entering an object through an operation with the inherited label simply retains its previous consistency type.

The combination of the consistency labels in the same object (or in the same thread) lead

to many interesting (as well as dangerous) execution time possibilities. Especially when s-threads update data being read/updated by gcp or lcp threads. The complete discussion of the semantics, behavior and implementation of this scheme is beyond the scope of this paper, and the reader is referred to [ChDa89].

4.2. Fault Tolerance

Transaction processing systems and systems like *Clouds* that support consistency of data, provide guarantees about consistency of data if computations do not complete (due to failures). However they do not guarantee success of computations. The following section discusses a subsystem for *Clouds* that is designed to allow non-stop computations.

This system uses a mechanism called *parallel execution threads* or PET. The attributes of the system guarantee uninterrupted processing in face of pre-existing (static) failures, as well as system and software failures that occur while a resilient computation is in progress (dynamic failures).

To obtain these property, the basic requirements of the system are:

- Replication of objects, for tolerating static and dynamic failures.
- Replication of computation, for tolerating dynamic failures.
- A quorum-based updating method and coordinated commit mechanism to make the scheme work.

The PET system works by first replicating all the critical objects at different sites in the system. The degree of replication is dependent on the degree of resilience required.

When a resilient computation is initiated, separate replicated actions (gcp-threads), on a number of sites. The number of sites is another parameter provided by the user, and reflects the degree of resilience required. The separate actions (or Parallel Execution Threads) run using DSM invocations on the replicated objects. An invocation by one

Page missing from report

provided by Argus is for the Argus language. *Clouds* on the other hand is a general purpose operating system, not tied to any language. Currently a modified version of C++ is being used for programming applications for *Clouds*, but otehr languages will be added later.

5.2. Eden

Eden is a object-based distributed system, implemented on the Unix operating system at the University of Washington. Eden objects (called Ejects) use the active object paradigm, that is each object consists of a process and an address space. An invocation of the object consists of sending a message to the (server) process in the object, which executes the requested routine, and returns the results in a reply [Alm83, AlBl83, NoPr85].

Since every object in the system needs to have a process servicing it, this could lead to too many processes. Thus Eden has an *active* and a *passive* representation of objects. The passive representation is the core image of the object stored on the disk. When an object is invoked, it must be active, thus invoking a passive object involves activating it. A process is created by 'exec'-ing the core image of the object (frozen earlier), and then performs the required operation. The activation of passive objects is an expensive operation. Also concurrent invocations of objects are difficult and are handled through multi-threaded processes or coroutines.

The active object paradigm and the Unix-based implementation are some of the major differences between Eden and *Clouds*. Eden also provides support for transaction and replication objects (called Replects). The transaction support and replication were added after the basic Eden system was designed and have some limitations due to way Unix handles disk I/O.

5.3. Cronus

Cronus is an operating system designed and implemented at BBN Laboratories. Some of the salient points of Cronus are the intergration of Cronus functions with Unix functions, the ability of Cronus to handle a wide variety of hardware and the coexistence of Cronus on

a distributed set of machines running Unix, as well as several other host operating systems [BeRe85, GuDe86, ScTh86].

Like Eden, Cronus uses the active objects. This is necessary to be able to make Cronus run on top of most host operating systems. Cronus objects are handled by managers. Often a single manager can handle several objects, by mapping the objects into its address space. The managers are servers and receive invocation requests through catalogued ports. Any Unix process on any machine on the network can avail of Cronus services from any manager, by sending a message to the appropriate manager. By using canonical data forms, the machine dependencies of data representations are made transparent. Irrespective of the machine types, any Unix machine can invoke Cronus objects in a location independent fashion.

5.4. ISIS

ISIS (version 1) is a distributed operating system, developed at Cornell University, to support fault tolerant computing. ISIS has been implemented on top of Unix. It uses replication and checkpointing to achieve failure resilience. If data object is declared to be k-resilient, the system creates k+1 copies of the object. The replicated object invocation is handled by invoking one replica and transmitting the state updates to all replicas. Checkpointing at each invocation is used to recover from failures [Bi85].

The goals and attributes of ISIS are different from *Clouds*. ISIS is built on top of some interesting communication primitives and is not built as a general purpose computing environment.

5.5. ArchOS and Alpha

Alpha is the kernel for the ArchOS operating system developed by the Archons project at Carnegie Mellon University. Like *Clouds*, the Alpha kernel is a native operating system kernel designed to run on the special hardware called Alpha-nodes. The Alpha kernel uses passive objects residing in their own virtual spaces, similar to *Clouds*. ArchOS is designed for real time applications support-

Page missing from report

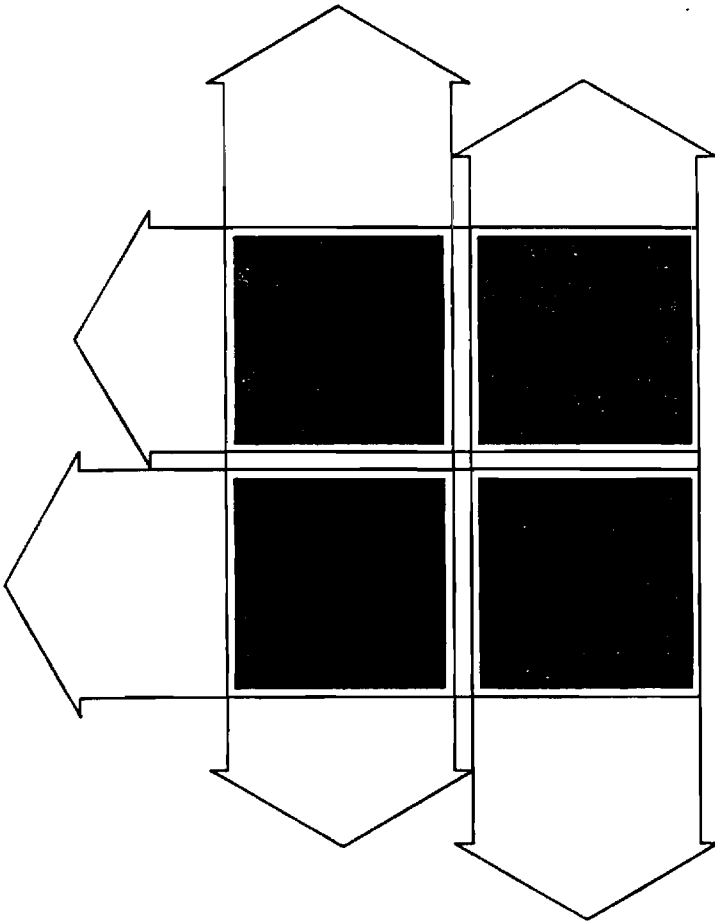
programming support research; Jose Bernabeu, Yousef Khalidi and Phil Hutto for their efforts in making the v.1. kernel usable and for the design and implementation of *Ra*; Sathis Menon for significant contribution to the implementation of v.2. as well as managing the software development effort. Also Mustaque Ahamad, Ray Chen, Greg Kenley, Kishore Ramachandran, Henry Strickland and Chris Wilkenloh for their participation in and contributions to the project.

8. References

- [Ac86] Accetta M, et. al. *Mach: A New Kernel Foundation for Unix Development*, Technical Report, Carnegie Mellon University.
- [AhAm87] M. Ahamad, M. Ammar, J. Bernabeu and M. Y. Khalidi, *A Multicast Scheme for Locating Objects in a Distributed System*. Technical Report GIT-ICS-87/01, School of Information and Computer Science, Georgia Tech, January 1987.
- [AhDa87] M. Ahamad and P. Dasgupta, *Parallel Execution Threads: An Approach to Atomic Actions*. Technical Report GIT-ICSD-87/16. School of Information and Computer Science, Georgia Tech.
- [AhDaLeWi87] M. Ahamad, P. Dasgupta, R.J. LeBlanc and C.T. Wilkes, *Fault Tolerant Computing in an Object-Based Distributed System*, Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems, Williamsburg, March 18-19, 1987.
- [AhDaLe88] M. Ahamad, P. Dasgupta and R.J. LeBlanc, *Atomic Replicated Computations in an Object-Based System*. In Preparation.
- [Alm83] G. T. Almes, *The Evolution of the Eden Invocation Mechanism*, Technical Report 83-01-03, Department of Computer Science, University of Washington, 1983.
- [Al83] J. E. Allchin, *An Architecture for Reliable Decentralized Systems*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, (Also released as technical report GIT-ICS-83/23,) 1983.
- [AlBl83] G. T. Almes, A. P. Black and E. D. Lazowska and J. D. Noe, *The Eden System: A Technical Review*, University of Washington Department of Computer Science, Technical Report 83-10-05 October 1983.
- [Be88] J. Bernabeu, *Location Finding Algorithm in Distributed Systems*, Ph.D. Thesis, School of Info. and Computer Science, Georgia Tech, December 1988.
- [BeHuKh88] J. M. Bernabeu Auban, P. W. Hutto and M. Y. A. Khalidi, M. Ahamad, W.F. Appelbe, P. Dasgupta, R.J. LeBlanc and U. Ramachandran. *Clouds - A Distributed Object-based Operating System: Architecture and Kernel Implementation*, European Unix Systems User Group Autumn Conference (EUUG), October 1988.
- [BeHuKh89] J. M. Bernabeu Auban, P. W. Hutto and M. Y. A. Khalidi, M. Ahamad, W.F. Appelbe, P. Dasgupta, R.J. LeBlanc and U. Ramachandran. *The Architecture of the Ra: A Kernel for Clouds*, Proceedings of the 22nd Hawaii International Conference on System Sciences, January 1989. [Also available as GIT-ICS-88/25].
- [BeRe85] J. C. Berets, R. A. Mucci and R. E. Schantz, *Cronus: A Testbed for Developing Distributed Systems*, October 1985 IEEE Communications Society, IEEE Military Communications Conference.
- [Bi85] K. P. Birman and others, *An Overview of the ISIS Project*, Distributed Processing Technical Committee Newsletter, IEEE Computer Society (7,2) October 1985 (Special issue on Reliable Distributed Systems).
- [Ch86a] D.R. Cheriton, *Problem-Oriented Shared Memory: A Decentralized Approach to Distributed Systems Design*, Proceedings of the 6th Distributed Computing Systems, May 1986.
- [Ch86b] D. R. Cheriton *VMTP: A Transport Protocol for the Next Generation of Communication Systems*, Proceedings of the SIGCOMM, 1986.
- [Ch88] D. Cheriton *The V Distributed System*. Communications of the ACM, March 1988.
- [ChDa89] R. Chen and P. Dasgupta, *Linking Consistency with Object/Thread Semantics: An Approach to Robust Computations*, 9th International Conference on Distributed Computing Systems, Newport Beach CA, June 5th-7th, 1989.
- [Da86] P. Dasgupta, *A Probe-Based Fault Tolerant Scheme for an Object-Based Operating System*, Proceeding of the 1st ACM Conference on Object Oriented Programming Systems, Languages and Applications. Portland OR. 1986.
- [DaLeAp88] P. Dasgupta, R.J. LeBlanc and W.F. Appelbe, *The Clouds Distributed Operating System: A functional description, related work and implementation details*, Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, June 1988.
- [Ge85] D. Gelernter, *Generating Communication in Linda*, ACM Transaction on Computer Systems, January 1985.
- [GuDe86] R. F. Gurwitz, M. A. Dean and R. E. Schantz, *Programming Support in the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.
- [GrSeWe86] I. Greif, R. Seliger and W. Weihl *Atomic Data Abstractions in a Distributed Collaborative*

Page missing from report

-
- [WiLe88] C.T. Wilkes and R.J. LeBlanc, *Distributed Locking: A Mechanism for Constructing Highly Available Objects*, Proceedings of the 7th Symposium on Reliable Distributed Systems, Columbus, October 1988.
- [Wu74] W. A. Wulf and others, *HYDRA: The Kernel of a Multiprocessor Operating System*, Communications of the ACM, (17,6) June 1974.
- [WuLe81] W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill, Inc., 1981.



The 9th International
Conference on
Distributed
Computing
Systems

IEEE Computer Society Press

Washington ● Los Alamitos ● Brussels ● Tokyo

Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation[†]

Raymond C. Chen

Partha Dasgupta

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

This paper presents an object/thread based paradigm that links data consistency with object/thread semantics. The paradigm can be used to achieve a wide range of consistency semantics from strict atomic transactions to standard process semantics. The paradigm supports three types of data consistency. Object programmers indicate the type of consistency desired on a per-operation basis and the system performs automatic concurrency control and recovery management to ensure that those consistency requirements are met. This allows programmers to customize consistency and recovery on a per-application basis without having to supply complicated, custom recovery management schemes.

The paradigm allows robust and non-robust computation to operate concurrently on the same data in a well-defined manner. The operating system need support only one vehicle of computation — the thread.

1 Introduction

The ability of a system to maintain consistent data in the face of hardware breakdowns, computation aborts, and other forms of failure, is loosely termed *robustness*. Robust computation transforms data from one consistent state to another in spite of failures, where the definition of consistent is application dependent. Thus robust systems may be termed "consistency-preserving" systems. In a general purpose (distributed) system, there is a need for both robust as well as non-robust computations and data, as many applications do not want to (or have to) pay the overhead costs needed in managing robust computations.

The most popular paradigm for robust computation are the *atomic transaction* based paradigms. The transaction paradigm is based on the *totality* concept — in principle, all the effects of a transaction are reflected in the *stable* (or *permanent*) state or none are. However, the transaction paradigm can be quite restrictive and most systems that provide transactions ([Al183] [LS83] [PN85] [SBD*84]) also provide "escape" mechanisms by which users may take advantage of application-specific semantics to increase concurrency and decrease overhead without sacrificing correctness. For example, *Argus* ([LDH*87]) allows atomic actions to touch non-atomic data, thus allowing actions to communicate. A paradigm developed for *Clouds v.1* provides recoverable and non-recoverable data segments, as well as custom locking and commit handling ([AM83] [Wil87]). These can be tailored for a variety of applications that cannot run as strict transactions. The *Avalon/Camelot* system ([HW87])

allows users to write their own commit routines to provide custom recovery for similar purposes. *Quicksilver* ([IIMSC88]) goes further, allowing customized commit protocols as well as commit and recovery routines. However, writing algorithms that correctly use these custom locking and recovery schemes is left to the application programmers. These algorithms can be quite complex and intricate. *Locus* ([WPLP85]) allows processes and actions to co-exist. However, a transaction locks the portions of the files it accesses and the locks apply to both processes and transactions. This preserves serializability for transactions but penalizes processes when a transaction is accessing the same data.

We have developed and are implementing a paradigm that we feel meets the following goals:

- Allows the object programmer to tailor consistency and concurrency in a manner appropriate to each application.
- Supports a range of consistency requirements from best-effort consistency to strict atomic transactions.
- Enables the operating system to handle concurrency and consistency issues in a transparent manner, without requiring the programmer to develop complex locking and recovery schemes tailored for each application.
- Presents one uniform world-view that encompasses both robust and non-robust computation, allows them to co-exist in the same system and if desired, concurrently access the same data with well-defined results.

This paradigm is designed for systems supporting an object/thread model of computation ([WCC*74] [Jon79] [Lis82] [LS83] [Al183] [ABLN85] [BMS85] [B*85]). The only vehicle of computation in this paradigm is the thread. Consistency and concurrency control are expressed through the semantics of objects, object invocation, and thread creation.

The paradigm is being implemented as part of the *Clouds v.2* operating system, a passive object-based general-purpose distributed system. *Clouds v.2* is built around the *Ra* kernel, an extendible, minimal kernel that provides light-weight processes, segment-based virtual memory management, short-term scheduling, and the ability to plug in system-level objects to perform additional operating system level functions ([BAHK*89]). *Ra* has been implemented on Sun 3/60 workstations, is operational and the *Clouds v.2* implementation effort is currently underway.

[†]This work was supported in part by NSF grant CCR-86-19886 and NASA grant NAG-1-430.

Section 2 of this paper presents the basic object/thread model. Sections 3-6 present the paradigm while sections 6 and 7 deal with issues arising due to concurrent execution.

2 Objects and Threads

An object is a long lived entity containing state (persistent data) and a set of operations that operate on this state. The object can be thought of as a virtual address space that is named and permanent. In fact, objects can be implemented as permanent virtual spaces ([DLA88]). This virtual space contains the data as well as the operators. The operations are called *entry-points* to the object.

Operations on objects are invoked by threads of execution (or *threads* for short). A *thread* is the carrier of execution in a distributed system much like a process is the carrier of execution on a centralized system. The entry of a thread into an object space is called *object invocation*. Object invocation allows parameters to be passed to the operation being invoked at the discretion of the object programmer. A thread begins its computation at an entry point in an object and flows through all objects (if any) invoked by this operation. Invoked objects may reside on any site in the distributed system. The object invocation terminates when the operation *completes* (returns). Note that an operation on an object can invoke operations on other objects (or operations on the same object). Objects are *passive* in the sense that they define the data and the operations, but do not execute by themselves. The threads are the active component of this programming model. This programming model is based on the processing environment provided by the *Clouds* operating system.

3 The Basics

Our approach is based on the notion of three types of consistency that an object programmer would want to maintain — global consistency, local consistency, and standard. Global consistency reflects a need on part of the object programmer to maintain a group of cooperating objects in a consistent manner. Properly used, global consistency guarantees that the operating system will automatically control concurrency and recovery across a set of objects so that their permanent states will stay both internally consistent and consistent with respect to each other.

Local consistency, on the other hand, is appropriate for those circumstances where *inter-object* consistency may be too strong a criterion. Sometimes, consistency within one object, or *intra-object* consistency is enough. Local consistency, properly used, guarantees that the system will control concurrency and recovery in a way that the object state will always stay internally consistent.

The standard degree of consistency is the degree of consistency that users in the process world have become accustomed to — that is to say no guarantees at all. If nothing fails, the data will be consistent, but the system can not guarantee consistency in the face of failure.

Object programmers label object operations. These labels indicate to the operating system the type of consistency desired by the object programmer for each operation. Threads carry consistency labels which indicate the type of consistency

Thread Label at invocation	Operation Consistency Label			
	GCP	LCP	S	I
GCP	No change	T → LCP	T → S	No change
LCP	T → GCP	T → LCP [†]	T → S	No change
S	T → GCP	T → LCP	No change	No change

Note: "T →" ≡ "Thread transforms to"

[†] Though the label remains the same, this is a transforming invocation.

Figure 1: Consistency Label Compatibility

the thread currently supports. When a thread invokes an object, the thread and operation labels are checked to see if they are compatible. If they are not, the thread *transforms* to meet the requirements of the object operation. The thread transforms back to its previous state when the operation terminates, whereupon the system performs the appropriate synchronization and recovery processing.

Objects contain persistent data items and reside in a single-level store backed by stable storage ([LS79]). The versions residing in the single-level store are known as the *base* (or *permanent*) *versions* while those residing on stable storage are known as the *stable versions*. Stable versions are presumed to survive system failures. No such assumption is made about the base versions.

4 Consistent Operations

Each entry-point is marked with one of four labels by the object programmer:

- Global Consistency Preserving (GCP)
- Local Consistency Preserving (LCP)
- Standard (S)
- Inherited (I)

Each thread of execution in the system also bears a consistency label. A thread may be a *global consistency preserving* thread (*gcp-thread*), *local consistency preserving* thread (*lcp-thread*), or a *standard* thread (*s-thread*) depending on whether the state of the thread's consistency label is set to global consistency preserving, local consistency preserving, or standard at that time. Both *gcp-threads* and *lcp-threads* reflect a commitment towards at least a minimal degree of data consistency and can be classified as different types of consistency-preserving threads (*cp-threads*). Thus in cases where the discussion applies to both *lcp-threads* and *gcp-threads*, we shall refer to both *gcp-threads* and *lcp-threads* as *cp-threads* and distinguish between them only when necessary.

When a thread with consistency label *X* invokes an operation *T* marked with consistency label *Y*, if *X* is not *compatible* with *Y*, then the thread *transforms* to a thread bearing consistency label *Y*. The consistency label of the thread reverts to the original value *X* when operation *T* completes. This process is called *thread transformation* and *T* is termed a *transforming invocation*. Thread label/operation label compatibility is summarized in figure 1.

4.1 Global Consistency Preserving

a thread invokes a *global consistency preserving* (gcp) operation and the invoking thread is not a gcp-thread, the thread becomes a gcp-thread and the current invocation is a *transforming invocation* for that thread.

When a gcp-thread attempts to update a persistent data item, the update is not immediately reflected in the permanent version of the data item. Instead, a new *shadow* version of that item is created. Further updates to that item by the same thread are reflected in the shadow version.

Let a *change-set* be the set of the names of all data items or which a gcp-thread has created a shadow version. A change-set is associated with each gcp transforming invocation. A data item is named by a change-set if the name of the data item is contained in the change-set. Every time a gcp-thread updates a persistent data item if the item is not named by the change-set of the transforming invocation for that gcp-thread, the name of the item being updated is added to that change-set. Upon completion of the transforming invocation, the change-set for that invocation will be atomically committed (or aborted) to the base and stable versions. A change-set is committed by atomically setting the values contained in both the base and stable versions of all items named by the change-set equal to the values contained in the shadow versions of all items named by the change-set. This normally requires executing a 2-phase commit protocol ([Gra79]). We refer to this process as committing changes or committing a change-set. A change-set is aborted by simply discarding the shadow versions of all data items named by the change-set.

If a gcp-thread invokes a gcp entry-point, the thread remains a gcp-thread and the operation is executed. Since the thread was already a gcp-thread when the current operation was invoked, the thread must have already invoked a transforming invocation. Therefore, updates made during the current operation are not committed when the current operation completes. The updates are reflected in the change-set of the transforming invocation and are committed (or aborted) when the transforming invocation commits (or aborts) its change-set. Since gcp-threads operate on shadow versions, no change made while the thread was a gcp-thread will be made permanent unless the transforming invocation successfully completes and all shadow versions are successfully committed.

4.2 Local Consistency Preserving

Local consistency preserving (lcp) operations are similar to gcp operations in that all changes made during an lcp invocation are made on new shadow versions and atomically committed (or aborted) when the lcp operation completes. However, every invocation of an lcp operation is a transforming invocation that transforms the invoking thread into an lcp-thread regardless of the previous state of the thread. That is, all changes made in an lcp invocation are committed or aborted when the invocation terminates even though the thread may have been executing as a gcp-thread before invoking the lcp operation.

Like gcp operations, each lcp invocation has a change-set associated with it and that change-set is committed (or aborted) when the invocation completes.

4.3 Standard

If a thread invoking a *standard* (s) operation is not an s-thread, then the thread is transformed into an s-thread and the current invocation is a transforming invocation. Upon completing the invocation, the thread reverts to its previous state.

Standard invocations do not commit or abort their changes, thus they do not have a change-set associated with them. If a thread updates the value of a data item in a standard invocation, that update does not create a shadow version. The update is instead applied directly to the latest version of the data item. The system will eventually propagate the changes to stable storage, however, the system makes no guarantees as to when this will occur. This is consistent with the "best-effort" semantics of s-threads and the notion of a process. (The definition of "latest version" will be discussed in greater detail in section 6.3.)

4.4 Inherited

Inherited operations are compatible with all threads and indicate the absence of a consistency requirement. Thus, inherited entry-points inherit the consistency label of the entering thread. If the invoking thread is a gcp-thread, the invocation behaves as a gcp invocation would. The same is true of s-threads or lcp-threads.

Inherited operations can be used to provide the object programmer with even more control over object consistency. Inherited operations can be used for operations that do not access persistent data or as "filters". Inherited entry-points provide a means by which objects may first examine the consistency label on the thread before allowing further processing to occur. Depending on the state of the thread and the object implementation, the object may allow the invocation to proceed (possibly invoking an internal entry-point to perform the actual processing) or it may reject the invocation by returning to the caller. This allows an object programmer to provide a uniform external interface for all threads while using different implementations for different thread types (gcp, lcp, or s).

5 Thread Creation

We allow threads to create other threads to execute object operations. The newly-created thread is said to be the *child* of the thread that created it (which in turn is referred to as the *parent* thread). A parent thread may monitor the status of its children, kill, suspend, and resume them. All threads execute concurrently.

In addition to consistency labels, all threads are also assigned a *nesting* attribute. Threads may be either *top-level* threads or *nested* threads. If the parent thread is a gcp-thread, the child thread may be created as either a top-level thread or a nested thread. Otherwise the child is created as a top-level thread.

Top-level threads are s-threads when first created. Nested threads are gcp-threads when created and the first operation invoked by the thread is a transforming invocation for that thread. If a nested gcp-thread transforms into a non-gcp thread, it is treated as a top-level thread until the invocation that transformed the thread into a non-gcp thread terminates.

Top-level threads commit their changes to permanent state. Nested threads commit their changes to their parent. That is, changes committed by a nested gcp-thread are not immediately reflected in the permanent system state. Instead, those changes

are passed onto the parent. This will be discussed in greater detail in section 6.

6 Locking and Visibility

Visibility becomes an issue when two or more threads concurrently executing within the same object attempt to read or write the same data item. Consider the read or write a *touch* on that item. In our paradigm, the operating system enforces certain system level locking rules on permanent data by requesting and releasing system-level locks on behalf of some threads that attempt to touch that data. These locking requests and the locks themselves are neither visible nor controllable by the user. This locking is 2-phase ([EGLT76]) by definition as will become obvious in the discussion below and is performed automatically by the operating system.

6.1 Consistency-Preserving Threads

Consistency-preserving threads automatically lock any data they touch. Read/Write locks are automatically requested by the operating system on behalf of the executing cp-thread when the thread attempts to read/write a data item.

The locking rules for interacting cp-threads are similar to the ones described in [Mos81] for nested actions. Thread creation may lead to a tree of one or more cp-threads, all with a common top-level ancestor. We call these thread trees consistency-preserving thread-trees.

Let T be a cp-thread in a cp-thread tree. A thread U is an ancestor of T if it lies on the path from T to the root of the thread-tree. This path includes T itself, thus a thread is always an ancestor of itself. A cp-thread T has a depth of n if and only if the path from T to the root of the tree is n nodes long where the length of the trivial path (a node to itself) is length 1. A *live* cp-thread is a non-terminated thread that has invoked but not yet completed at least one transforming consistency-preserving invocation. A cp-thread T may write-lock permanent data only if all other live cp-threads that hold a read or write-lock on that data are ancestors of T . A cp-thread T may read-lock permanent data only if all other live cp-threads that hold a write-lock on that data are also ancestors of T . A cp-thread can not read (or write) a data item unless it obtains a read-lock (or write-lock) on that item. If a cp-thread attempts to read (or write) a permanent data item and the system can not grant that thread the read (or write) lock due to the locking rules, the thread blocks until the lock can be granted.

In addition, to prevent undesirable interactions between concurrently executing parents and children on a cp-thread tree, a thread may update permanent state only if it has no live descendants. This constraint ensures that all non-terminated children of a thread will see the same (intermediate) versions of that thread's changes (if any) to permanent state.

Any locks held by a nested cp-thread are propagated to its parent if the thread successfully commits or released if the thread aborts. (Aborts are presumed to return only a failure code to the caller.) Locks held by top-level cp-threads are released when the transforming invocation commits or aborts. These are the *only* situations where system-level locks are released. Aborting a cp-thread in a thread-tree automatically aborts all its descendants.

6.2 Version Stacks

The cp-thread locking rules outlined in section 6.1 are a straightforward adaptation of the rules laid out for nested actions in [Mos81]. These rules lead to modeling uncommitted changes to a data item by versions on a *version stack*.

Every persistent data item can be viewed as possessing a version stack with the permanent version at the base. The entire stack exists in volatile memory, however, the permanent versions are backed by *stable versions* residing on stable storage and all commits to permanent versions are mirrored on the stable versions.

The versions on the version stack are referred to as versions $0, 1, 2, \dots, n-1, n$ where version 0 is the permanent version, version n is the top version and n is the height (or depth) of the stack. Version creation for gcp-threads is based on the nesting level of the thread in the thread-tree. Version creation for lcp-threads is based on the number of times different unfinished lcp invocations in the same object have touched the same item. In both cases, the idea behind the version management is to allow gcp and lcp invocations to operate on private shadow versions and then atomically commit or abort those versions.

When a gcp-thread T of depth d gains a write-lock on a data item it has not previously touched, a new version of that item (call it X) is created and initialized to the value of the version on top of the version stack. Let n be the depth of the version stack. If $d - n > 1$, then $d - n - 1$ new versions are also created, initialized to the value of the top version on the stack, and placed onto the version stack as versions $n + 1, n + 2, \dots, d - 1$. X is then placed on top of the stack as version d . If $d - n = 1$, X is simply placed on top of the version stack as version d .

When an lcp-thread touches a data item it has not touched in the current lcp invocation, a new version is created, initialized to the value of the top-most version on the version stack, and placed on the stack.

The locking rules together with the rule that a thread may update permanent state only if it has no live descendants ensure that there exists only one version at each level of the stack. Thus the stack is always a *linear stack* and never a *cactus stack*.

When a cp-thread commits, all items in its change-set are committed or aborted. A gcp-thread of depth n commits a change by copying the top version of the item on the version stack (version n) to the next-most-recent version on the version stack (version $n - 1$) and then discarding the top version. Thus nested gcp-threads commit their changes to intermediate versions that will then be committed or aborted by their parents. However, having a depth of one, top-level gcp-threads commit their changes to the permanent versions.

A gcp-thread tree may create a version stack with a depth of greater than one if non-root threads update data. An lcp-thread may also create a version stack with a depth of greater than one through recursive invocations. However the semantics of local consistency demand that the effects of an lcp invocation be permanent if the invocation completes successfully. Therefore, an lcp-thread commits a change by copying the top version on the stack to all other versions on that stack and then discarding the top version. In effect, an lcp-thread commit is a *write-through* commit.

Instantaneous View (in memory)

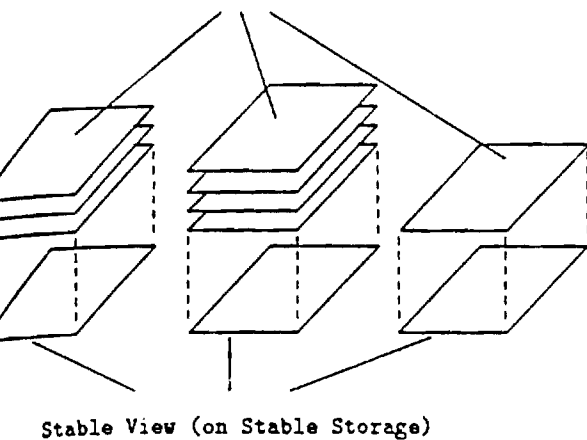


Figure 2: Instantaneous and Stable Views

All cp-threads abort a change to an item by discarding the top version of the version stack for that item.

3 Views

The locking rules in section 6.1 are designed to allow only one s-thread to hold a write-lock on a data item and update it without blocking. However, these updates affect only the top-most version on the version stack. It follows that for cp-threads, the top-most version on the stack can be thought of as the *latest* version since it contains the latest update and is the only version that can be read or written by a cp-thread.

s-threads, however, may operate directly on the permanent version of a data item. Assuming for the moment that no cp-thread is working with data items used by s-threads, the permanent versions being updated by s-threads are also the top-most versions on the version stacks for those items.

This leads to two appealing global views of the system. One is the *stable view* consisting of the stable versions of the persistent state. This view would be present even if the system were to crash that instant and then recover. Another view is the *instantaneous view*. This view consists of the latest version of each persistent data item, which corresponds to the top version on the version stack for every persistent data item whether that version is a shadow version or a permanent version. This instantaneous view may not be valid in the case of failures. However, it is an intuitively appealing view in that, at the instant the system is examined, it is an accurate snapshot of the latest changes made to the system (see figure 2).

Thread Interactions

In our paradigm, s-threads always operate on the instantaneous view. Changes made by s-threads become immediately visible to other s-threads since the change is reflected in the instantaneous view. If the changes are made to a permanent version, following a system failure, those changes will eventually be propagated to the stable view. This is consistent with the usual semantics.

cp-thread trees operate on the instantaneous view and alter the stable view by committing portions of the instantaneous view to the stable view. This allows programmers to control consistency in the stable view and is consistent with the usual notion of a transaction. The change-set of a transforming invocation for a top-level cp-thread at commit time can be regarded as the set of data items that have to be committed to bring the stable view into agreement with the portion of the instantaneous view that has been altered as a result of that invocation. However, while the locking and visibility rules defined in section 6.1 regulate the interactions between cp-threads, they do not address the situations that may arise when s-threads and cp-threads attempt to operate concurrently on the same persistent data.

For in our paradigm, standard threads do not acquire system-level locks. We feel that this would be too restrictive. Forcing s-threads to acquire system-level locks would be treating them like *de facto* transactions. This seems redundant given that a programmer can prevent s-threads from interacting with cp-threads inside an object by using inherited entry-points or lcp/gcp operation labels. If the programmer wishes to use s-threads in an object and synchronize their activity, user-level locking or some other form of explicit concurrency control may be used. This is again consistent with the process paradigm. However, this lack of full automatic synchronization leads to three situations that must be defined: s-threads reading writes made by cp-thread trees, cp-threads reading writes made by s-threads, and s-threads overwriting updates made by cp-thread trees.

7.1 Standard Reads of Consistent Writes

s-threads do not acquire system-level locks, hence they will never be blocked by the system when accessing permanent state. Since s-threads see the instantaneous view of the system, they see the latest versions of all data *including* changes made, but not committed, by cp-threads. Thus, if an s-thread attempts to read data touched by a cp-thread, the s-thread will not block and will read the latest (instantaneous) version of the data.

7.2 Consistent Reads of Standard Writes

S-threads may write to data that have been read-locked by a cp-thread. The write will update the latest (instantaneous) version of the data. A cp-thread commit or abort affects only versions of data items *created* by that cp-thread and its committed descendants and placed on version stacks. Reads by a cp-thread do not create new versions on the version stack. If a cp-thread and/or its committed descendants read a data item updated by an s-thread and neither the cp-thread nor any of its descendants updates that item, the commit processing for that cp-thread will not include the update made by the s-thread.

7.3 Standard Writes and Consistent Writes

We allow s-threads to overwrite uncommitted updates made by cp-threads. If this happens, the s-thread overwrite does not invalidate the system-level locks on the overwritten data. The overwrite appears in the instantaneous view and will be committed or aborted when that item is committed or aborted by the concurrently executing cp-thread *exactly* as if the update had been made by the cp-thread. Thus, in this case, the permanence of the s-thread overwrite is in doubt and is governed by the commit processing of the cp-thread holding the system-

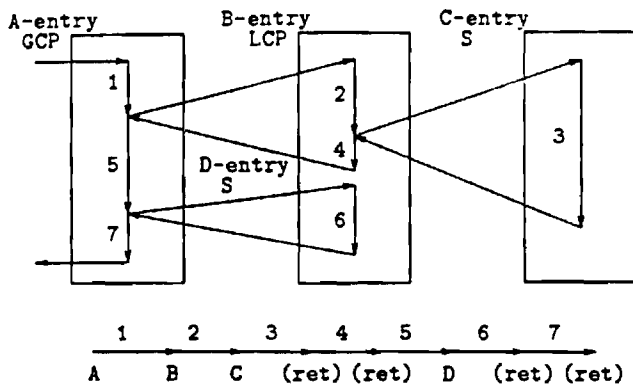


Figure 3: A Thread of Execution

level write-lock on the overwritten data item. This, however is consistent with the "best-effort" semantics of *s*-threads.

7.4 Mutable Threads

Figure 3 shows an example of a *gcp*-thread invoking a number of object operations. Operation *A* (in object #1) is a *gcp* entry-point. Operation *B* (in object #2) is an *lcp* entry-point. Operations *C* and *D* (in object #3 and #2 respectively) are standard entry-points. In this example, all the invocations except the invocation of operation *A* (or invocation *A*) are transforming invocations. The changes made in segments #1, #5, and #7 while the thread is a *gcp*-thread are visible in the instantaneous view as soon as they are made. However, the affected items are system-locked and the changes will not be committed or aborted until the transforming invocation (not shown) that made the thread a *gcp*-thread completes. Changes made in thread segment #2 and #4 are simultaneously committed to the stable view or aborted when invocation *B* completes. Changes made in thread segment #3 (invocation *C*) are performed without system-locking. Thus, those changes are immediately visible in the instantaneous view regardless of the state of invocation *B*. They may eventually be committed to the stable view by a *cp*-thread or committed by the system or aborted. However they will not be committed or aborted as a direct result of any of the invocations shown in figure 3. Likewise, changes made in thread segment #6 become visible in the instantaneous view without affecting system-level locks and the changes will eventually be committed or aborted (but not as a direct result of any of the invocations shown in figure 3).

8 Comments

Given this set of thread creation and consistency semantics, we can support the functionality of top-level and nested actions. If a standard thread invokes a *gcp* operation, the thread is transformed into a *gcp*-thread for the duration of the object invocation and becomes the equivalent of a top-level action until the operation completes or invokes an *lcp* or standard operation. A top-level action can be also created by creating a top-level thread to invoke a *gcp* operation. A nested action can be created by creating a nested thread to invoke a *gcp* object operation.

If a set of objects are programmed so that only *cp*-threads are allowed to execute object operations, all operations will be se-

rializable and the above-mentioned actions will behave as strict atomic transactions. If the objects are programmed so that only *cp*-threads may update permanent data or so that *cp*-threads do not read updates made by *s*-threads (if any), then every "action" (*cp*-thread tree) will be serializable with respect to every other *cp*-thread tree.

Serializability breaks down only if *s*-threads are allowed to view a *cp*-thread's intermediate results and make their own results visible to other *cp*-threads. However, each object has full control over the thread interactions that may occur within it. Although interactions between *s*-threads and *cp*-threads are not controlled by the system, they are under the control of the object programmer on a per-object basis.

Thus this paradigm allows for consistent updates to permanent object state in accordance with the view/failure atomicity paradigm and nested action semantics per [Mos81]. Interesting and complex semantics may also be achieved by using well-defined system properties that in other systems would have to be programmed in a completely custom manner. The system properties may also be supplemented by using user-level concurrency control.

9 A Robust Object File System

Robust file objects on a disk system can be defined as in figure 4. These robust file objects may be read from and written to. The *read* and *write* operations are *lcp* but appear to behave like atomic reads and writes to users of the object. Reads will always read consistent data and writes will not leave the object in an inconsistent state. The *truncate* operation (not shown) truncates the file at a specified block. The *segment* directives form system-locking segments (see section 10). Parameters are assumed to be in/out by value.

The data in a file object are contained in a number of fixed size "virtual" blocks. Before being used to hold data, virtual blocks must be backed by real storage blocks allocated by the disk block allocator. If a file needs to grow (caused by a write at the end of the file), all file objects call the disk block allocator object. The block allocator manages a disk map, allocation statistics, and keeps track of which object has been allocated what blocks. This information is all bundled into one system-locking segment.

The block allocator has three *lcp*-entry points: *get_block*, *release_block*, and *garbage_collect*, and a standard entry-point, *get_status*. The *get_status* operation is a non-blocking, read-only operation that exploits an *s*-thread's ability to read system-locked data without blocking. The operation reads the variables containing the allocation statistics which allows users to quickly obtain the approximate state of the allocator. The *get_block* operation allocates a backing storage block for a block in the object. The *release_block* operation releases all backing blocks beyond a specified block effectively truncating the file at that block. Both *get_block* and *release_block* update the allocation statistics to reflect the blocks allocated/released. Since *get_block* and *release_block* are *lcp* operations, and *get_status* is a read-only operation, the block allocator's internal data structures always appear to be in a consistent state.

A write call may append more than one block at a time, thus it is possible for a number of blocks to be allocated to a file ob-

```

block {
char data[BLOCKSIZE];
int block_id;
}

file {
segment file_data {
int numblocks;    ///backed by allocator
block filedata[] ///real data;
}
}

segment block_data {
//list of integers
}

lcp-entry status read(int,
char [BLOCKSIZE]);
lcp-entry status write(int,
char [BLOCKSIZE]);
lcp-entry status truncate(int);
lcp-entry status get_block_ids(id_list ids);

try status
int position, int nblocks, char buf[nblocks*BLOCKSIZE])
for (int i = position; i < nblocks + position; i++) {
if (i >= numblocks)
if (int block_id = get_block(i)) {
numblocks++;
// then add block_id to linked
// list of blocks_ids
} else abort();

strcpy(data[i], buf, BLOCKSIZE);
}

return(SUCCESS);
}

```

- 1) determine which segment was being accessed
- 2) if the segment is not a data segment, jump to the real access violation handler
- 3) if access was a read
 - a) check to see if a read lock is held on that segment
 - b) if not, get read lock (call system-lock manager)
 - c) set page table entries for the segment to allow reads
 - d) mark read lock held on the segment
 - e) return from interrupt
- 4) if write lock not held, get write-lock (call system-lock manager)
- 5) if thread is a gcp-thread of depth n and the version stack is of depth n, set the pte to allow write and return from interrupt
- 6) mark page in segment as shadowed, if not already marked
- 7) preserve current version by copying the page
- 8) push the version-stack record of the page (page #, thread depth, retrieval key) onto the version stack for that page
- 9) set protection mask on pte to allow write access
- 10) return from interrupt

Figure 5: An Access Violation Handler

```

try status
int position, int nblocks, char buf[nblocks*BLOCKSIZE])

if (position + nblocks > numblocks)
return(FAILURE);

for (int i = position; i < position + nblocks; i++)
strcpy(data[i], buf, BLOCKSIZE);

return(SUCCESS);
}

```

garbage that has to be collected. However, since commits should happen much more frequently than aborts, this should be a good trade-off. This example demonstrates how programmers using this paradigm can use a consistent set of object/thread semantics to make application-specific tradeoffs concerning concurrency, resilience, efficiency, fairness, and implementation complexity.

10 Implementation

The semantics presented here, while abstract, naturally lend themselves to being implemented in and supported by the operating system. The semantics are a super-set of the basic object/thread semantics as defined in section 2, and certain features such as the automatic system-level locking would be difficult and/or prohibitively expensive to implement at a language level.

This paradigm is being implemented as part of the Clouds v.2 operating system. User-level objects, object invocation, commit/abort processing, and system-level locking are being implemented as system-level objects in the Ra kernel. User objects are composed of Ra segments that contain either code or data. System-level locking is performed on *locking segments*. A locking segment may contain one or more (possibly discontinuous) Ra segments which may contain arbitrary variables or data structures. This allows flexible grouping of variables and data structures into logical locking groups.

The algorithms for handling nested and top-level cp-thread commits are adapted from nested and top-level action commit algorithms. However, since the system-level locking applies to persistent data residing in a single-level store (memory backed by secondary storage), system-level locking and version-stack management are being implemented using the virtual memory system and protection mechanisms. If a cp-thread does not hold a write (read) lock on a segment, the thread's page table entries (pte's) for that segment will be set to prohibit write (read or write) access. Attempts to access read/write or write-protected

Figure 4: Partial Definition of Robust File Object

uring a write operation, only to have that write operation This would result in storage blocks allocated to an object not used. The garbage collection entry-point can be called locally to reclaim these blocks. The garbage collector ref from each object the list of blocks used by that object (*get_block_ids*). Blocks allocated on the disk map but not by any object can be reclaimed.

is implementation of file objects has atomic read/write tics and trades off garbage collection for increased con- cy, fairness, and faster commit processing. Commit pro- g is faster since all commits affect only one object. Concur- and fairness are increased since the block allocator object er locked by any one thread for very long. The price we that aborts become more expensive because they result in

data results an access violation, whereupon control passes to the operating system access violation handler (see figure 5). The access violation handler then determines if the thread should be blocked or allowed to proceed, whether a shadow version must be created, and which page table entries should be reset to allow read or write access. A simple access violation handler is shown in figure 5. This version shadows only the updated pages in a system-locking segment. A more sophisticated version could attempt to decrease the number of access violations by shadowing other pages around a touched page in anticipation of them being touched later.

1 Conclusions

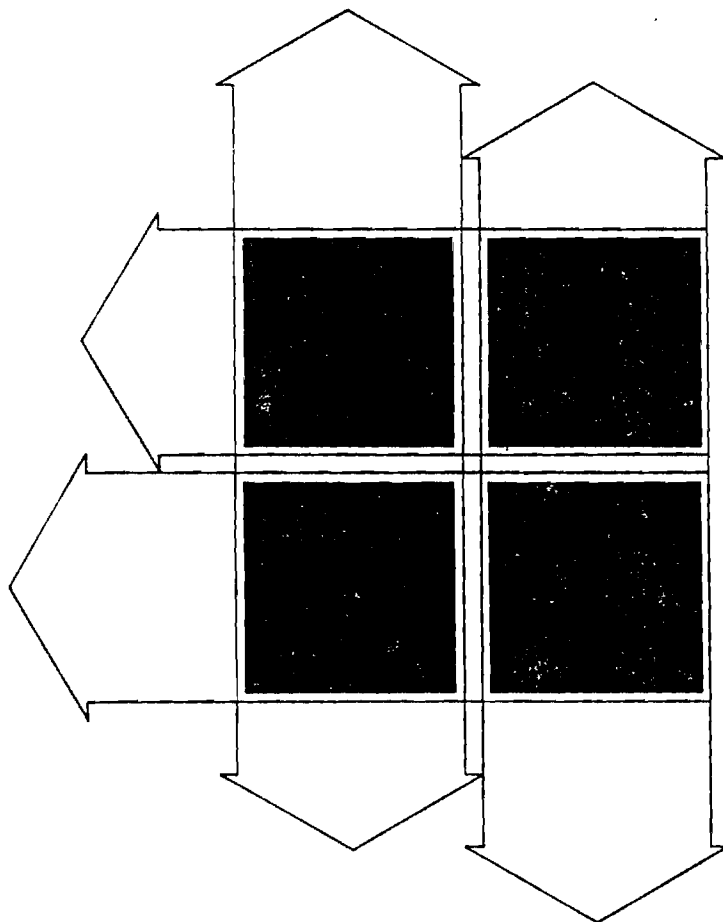
This paper presents an object-based paradigm which supports a wide range of robust programming and has but one computation abstraction — the thread. Consistency labeling mechanisms and thread creation semantics may be used to achieve action/nested-action semantics. However, thread transformation can be used to achieve threads of execution that do not behave like actions that are globally consistent in certain segments, non-consistent (standard) in others, and locally consistent in others without having to supply complicated custom recovery schemes. This paradigm enables operating system designers to support robust and non-robust computation in a uniform manner while also giving object programmers fine control over the degree of consistency maintained within their objects and the methods used to achieve that consistency.

2 Acknowledgements

The example in section 9 resulted from a discussion with Robert McCurley. We also wish to thank Bill Appelbe, Glenn Benson, Mich LeBlanc, and the other members of the Distributed Systems Group at Georgia Tech for their feedback and suggestions.

References

- [BLN85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43-59, January 1985.
- [All83] J. E. Allchin. *An Architecture for Reliable Decentralized Systems*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983.
- [AM83] J. E. Allchin and M. S. McKendry. Synchronization and Recovery of Actions. In *Proceedings of the Second ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, ACM, Montreal, August 1983.
- [Bir85] K. P. Birman et al. An Overview of the ISIS Project. *Distributed Processing Technical Committee Newsletter, IEEE Computer Society*, 7(2), October 1985. (Special issue on Reliable Distributed Systems).
- [AHK*89] Jose M. Bernabéu Aubán, Phillip W. Hutto, M. Yousef A. Khalidi, Mustaque Ahmad, William F. Appelbe, Partha Dasgupta, Richard J. LeBlanc, and Umakishore Ramachandran. The Architecture of Ra: A Kernel for Clouds. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.
- [MS85] J. C. Berets, R. A. Mucci, and R. E. Schantz. Cronus: A Tentbed for Developing Distributed Systems. In *IEEE Military Communications Conference*, IEEE Communications Society, October 1985.
- [DLA86] P. Dasgupta, R. LeBlanc, and W. Appelbe. The Clouds Distributed Operating System. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.
- [EGLI76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM*, 19(11):624-633, November 1976.
- [Gra79] James N. Gray. Notes On Database Operating Systems. In *Operating Systems: An Advanced Course*, Springer-Verlag, Berlin, 1979.
- [HMSC88] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82-106, February 1988.
- [HW87] M. P. Herlihy and J. M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, July 1987.
- [Jon79] A. K. Jones. The Object Model: A Conceptual Tool for Structuring Software. In *Operating Systems: An Advanced Course*, pages 7-16, Springer-Verlag, NY, 1979.
- [LDH*87] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. *Argus Reference Manual*. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1987.
- [Lis82] B. Liskov. On Linguistic Support for Distributed Programs. *IEEE Transactions on Software Engineering*, SE-8(3):203-210, May 1982.
- [LS79] B. W. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Storage System. 1979. Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, CA.
- [LS83] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381-404, July 1983.
- [Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1981.
- [PN85] C. Pu and J. D. Noe. *Nested Transactions for General Objects: The Eden Implementation*. Technical Report 85-12-03, Department of Computer Science, University of Washington, Seattle, WA, December 1985.
- [SBD*84] A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya, and P. M. Schwarz. Support for Distributed Transactions in the TABS Prototype. In *Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems*, pages 184-206, October 1984.
- [WCC*74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337-345, June 1974.
- [Wil87] C. T. Wilkes. *Programming Methodologies for Resilience and Availability*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.
- [WPLP85] Matthew J. Weinstein, Thomas W. Page, Jr., Brian K. Livezey, and Gerald J. Popek. Transactions and Synchronization in a Distributed Operating System. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 115-126, December 1985.



The 8th International
Conference on
Distributed
Computing
Systems

San Jose, California
June 13-17, 1988

SPONSORED BY

 **THE COMPUTER SOCIETY**
Technical Committee
on Distributed Processing

 **THE INSTITUTE OF ELECTRICAL
AND ELECTRONICS ENGINEERS, INC.**
IEEE

Computer Society Order Number 865
Library of Congress Number 87-83544
IEEE Catalog Number 86CH2541-1
ISBN 0-8186-0865-X

 **THE COMPUTER SOCIETY**



IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

**COMPUTER
SOCIETY
PRESS** 

The Clouds Distributed Operating System: †

Functional Description, Implementation Details and Related Work.

Partha Dasgupta, Richard J. LeBlanc Jr., & William F. Appelbe.

School of Information and Computer Science
Georgia Institute of Technology
Atlanta GA 30332

Abstract

Clouds is an operating system in a novel class of distributed operating systems providing the integration, reliability and structure that makes a distributed system usable. *Clouds* is designed to run on a set of general purpose computers that are connected via a medium-to-high speed local area network. The structure of *Clouds* promotes transparency, support for advanced programming paradigms, and integration of resource management, as well as a fair degree of autonomy at each site.

The system structuring paradigm chosen for the *Clouds* operating system, after substantial research, is an object/thread model. All instances of services, programs and data in *Clouds* are encapsulated in objects. The concept of persistent objects does away with the need for file systems, and replaces it with a more powerful concept, namely the object system. The facilities in *Clouds* include integration of resources through location transparency; support for various types of atomic operations, including conventional transactions; advanced support for achieving fault tolerance, and provisions for dynamic reconfiguration.

1. Introduction

Clouds is a distributed operating system under development. The goal of the *Clouds* project is to develop an instance of a class of distributed operating systems that provide the integration, reliability and structure necessary to make distributed computing system usable.

Clouds is designed to run on a set of general purpose computers (uniprocessors or multiprocessors) that are connected via a medium-to-high speed local area network. The major design objectives for *Clouds* are:

- Integration of resources through cooperation and location transparency.
- Usable support for various forms of atomicity, including transaction processing, and the ability to achieve fault tolerance (if needed).

† This research was partially supported by NASA under contract number NAG-1-430 and NSF under contract number DCS-8316590 and CCR-8619886.

- Efficient design and implementation.
- Simple and uniform interfaces for distributed processing.

The paradigm used for defining and implementing the software structure of the *Clouds* system, chosen after substantial research is an object/thread model. This model provides threads to support computation and objects to support an abstraction of storage. (These concepts are defined in sections 2 through 4). This model has been augmented to support atomicity of computation to provide support for reliable programs [Al83, ChDa87]. In this paper, we provide a functional description of the system (sections 2 to 6), some implementation details (section 7), and discussion of related work (section 9).

1.1. Current Status

The first version of the *Clouds* operating system has been implemented and is operational. This version is referred to *Clouds* v.1. This is being used as an experimental testbed by the implementors.

Some of the performance figures for *Clouds* v.1. were:

Local Invocations	10 msec
Remote Invocations	40 msec
Commit of 1 page data	180 msec

These figures are large due to several factors. The VAX architecture was not very suitable for implementing objects, and flushing of the translation buffers for each invocation causes the local invocation to be more expensive than expected [RaKh88]. The Ethernet hardware used in our VAX-11/750 is slow, and coupled with a non-optimized driver gives us poor performance on round trip messages and hence large remote invocation times. The disk used in the commit tests was also exceedingly slow (40msec seek, 25msec/page write.) However, the experience with this version has taught us that the approach works. It also taught us how to do it better.

The lessons learned from this implementation are being used to redesign the kernel and build a new version. The basic system paradigm, the semantics of objects and threads and the goals of the project remain unchanged and v.2. will be identical to v.1. in this respect.

The structure of *Clouds* v.2. is different. The operating system will consist of a minimal kernel called "Ra". Ra will support the basic function of the system, that is location independent object invocation. The operating system will be built on top of the Ra kernel using system level objects to provide systems services (user object management, synchronization, naming, atomicity and so on.)

2. Objects

All data, programs, devices and resources on *Clouds* are encapsulated in entities called objects. The only entity recognized by the system, other than an object is a thread. A *Clouds* object, at the lowest level of conception, is a virtual address space. Unlike virtual address spaces in conventional operating systems, a *Clouds* object is neither tied to any process nor is volatile. A *Clouds* object exists forever (like a file) unless explicitly deleted. As will be obvious in the following description of objects, *Clouds* objects are somewhat 'heavyweight', that is they are suited for storage and execution of large-grained data and programs. This is due to the fact that invocation and storage of objects bear some non-trivial overhead.

Every *Clouds* object is named. The name of an object, also known as its *capability*, is unique over the entire distributed system and does not include the location of the object. That is, the capability-based naming scheme in *Clouds* creates a uniform, flat system name space for objects, and allow for object mobility needed for load balancing and reconfiguration.

An object consists of a named address space, and the contents of the address space. Since it does not contain a process, it is completely passive. Hence, unlike objects in some object based systems, a *Clouds* object is not associated with any server process. (The first system to use passive objects, though in a multiprocessor system was Hydra [Wu74, WuLc81]).

Threads are the active entities in the system, and are used to execute the code in an object (details in sections 2 and 3). A thread executes in an object by entering it through one of several entry points, and after the execution is complete the thread leaves the object. Several threads can simultaneously enter an object and execute concurrently (or in parallel, if the host machine is a multiprocessor.)

Objects have structure. They contain, minimally, a code segment, a data segment and a mechanism for extending limits of storage allocated to the object. Threads that enter an object execute in the code segment. The data segment is accessible by the code in the code segment, but not by any other object. Thus the object has a wall around it which has some well-defined gateways, through which activity can come in. Data cannot be transmitted in or out of the object freely, but can be moved as parameters to the code segment entry points (see discussion on threads).

Clouds objects can be defined by the user or defined by the system. Most objects are user-defined. Some examples of system-defined objects are device drivers, name-service handlers, communication systems, systems software, utilities, and so on. The basic kernel (*Ra*) is not an object; it is an entity that provides the support for object invocation. A complete *Clouds* object can contain user-defined code and data; system-defined code and data that handle synchronization, recovery and commit; a volatile heap for temporary memory allocation; a permanent heap for allocating memory that will remain permanent as a part of the data structures in the object; locks; and capabilities to other objects.

Files in conventional systems can be conceived of a special case of a *Clouds* object. Thus, *Clouds* need not support a file system, but uses an object system. This is discussed in further detail in section 4.

Though, *Clouds* objects can be created, deleted and manipulated individually, the operating system is designed to support a class and instantiation mechanism. An object in the system

can be an instance of its *template*. An object of a certain type is created by invoking a 'create' operation on the template of this type. Each template is created by invoking a create operation on a single template-template, which can create any template, if provided, as argument, the code and data definitions of the template. The templates, the template-template and all the instances thereof, are regular *Clouds* objects, and, as discussed earlier, they exist from the time of creation, until explicitly deleted.

3. Threads

The only form of activity in the *Clouds* system is the thread. A thread can be viewed as a thread of control that executes code in objects, traversing objects as it executes. Threads can span objects, and can span machine boundaries. In fact, machine boundaries are invisible to the thread (and hence to the user). Threads are implemented in the *Clouds* system as lightweight processes, comprising of a PCB and a stack (but no virtual space). A thread that spans machine boundaries is implemented by several processes, one per site.

Upon creation, a thread starts up at an entry point of an object. As the thread executes, it executes code inside an object and manipulates the data inside this object. The code in the object can contain a procedure call to an operation of another object. When a thread executes this call, it temporarily leaves the caller object and enters the called object, and commences execution there. The thread returns to the caller object after the execution in the called object terminates. The calls to the entry point of objects are called *object invocations*. Object invocations can be nested. The code that is accessible by each entry point is known as an *operation* of the object.

A thread executes by processing operations defined inside many objects. Unlike processes in conventional operating systems, the thread often cross boundaries of virtual address spaces. Addressing in an address space is, however, limited to that address space, and thus the thread cannot access any data outside an address space. Control transfer between address spaces occurs through object invocation, and data transfer between address spaces occurs through parameters to object invocation.

When a thread executing in an object (or address space) executes a call to another object, it can provide the called operation with arguments. When the called operation terminates, it can send back result arguments. That is, object invocations may carry parameters in either direction.

These arguments are strictly data, they may not be addresses. Note that names (capabilities) are data. This restriction is necessary as the address space of each object are disjoint, and an address is meaningful only in the context of the appropriate object. Parameter passing uses the copy-in-copy-out method.

4. The Object/Thread Paradigm

The structure created by a system composed of objects and threads has several interesting properties.

First, all inter-object interfaces are procedural. Object invocations are equivalent to procedure calls on modules not sharing global data. The modules are permanent. The procedure calls work across machine boundaries. (Since the objects exists in a global name space, there is no user-level concept of machine boundaries.) Although local invocations and remote invocations (also known as remote procedure calls or RPC) are differentiated by the operating system, this is transparent to the applications and systems programmers.

Second, the storage mechanism used in the object-based world is quite different from that used in the conventional operating systems. Conventionally, the file is the storage medium of choice for data that has to persist, especially since memory is tied to processes and processes can die and lose all the contents of their memory. However, memory is easier to manage, more suited for structuring data and essential for processing. The object concept merges these two views of storage, and creates the permanent virtual space.

For instance, a conventional file is a special case of an object. That is, a file is an object with operations such as read, write, seek, and so on, defined in it. These operations transport data in and out of the object through parameters provided to the calls.

Though files can be implemented using objects, the need for having files disappear in most situations. Programs do not need to store data in file-like entities, since they can keep the data in the data space in each object, structure appropriately. The need for user-level naming of files transforms to the need for user-level naming for objects.

Just as *Clouds* does not have files, it does not provide user-level support for file (or disk) I/O. In fact there is no concept of a "disk" or such I/O devices (except user terminals). The system creates the illusion of a huge virtual memory space that is permanent (non-volatile), and thus the need for using disk storage from a programmer's point of view, is eliminated.

Messages are a paradigm of choice in message-based distributed systems. In the object-thread paradigm, like the need for I/O, the need for messages is eliminated. Threads need not communicate through messages. Thus ports are not supported. This allows a simplified system management strategy as the system does not have to maintain linkage information between threads and ports.

Just as files can be simulated for those in need for them, messages and ports can be easily simulated by an object consisting of a bounded buffer that implements the send and receive operations on the buffer. However, we feel that the need for files and messages are the product of the programming paradigms designed for systems supporting these features, and these are not necessary structuring tools for programming environments.

A programmer's view of the computing environment created by *Clouds* is apparent. It is a simple world of named address spaces (or objects). These objects live in computing systems on a LAN, but the machine boundaries are made transparent, creating a unified object space. Activity is provided by threads moving around amongst the population of objects through invocation; and data flow is implemented by parameter passing. The system thus looks like a set of *permanent address spaces* which support control flow through them, constituting what we term *object memory*.

This view of a distributed system does have some pitfalls. However these problems can be dealt with using simple techniques (implemented by the system), which are outlined below.

Threads aborting due to errors will leave permanent faulty data in objects they have modified. Failure of computers will result in similar mishaps. Multiple threads invoking the same object will cause errors due to race conditions and conflicts. More involved consistency violations may be the results of non-serializable executions. In a large distributed system, having thousands of objects and dozens of machines, corruption due to failure cannot be tolerated or easily repaired. The prevention of

such situations is achieved through the use of atomicity at the processing level (not necessarily atomic actions). The following section gives a brief overview of the atomicity properties supported by *Clouds*.

5. Atomicity

The action support is an area where the *Clouds* v.1. and *Clouds* v.2. differ.

5.1. Actions in *Clouds* v.1.

In the first design, *Clouds* supported atomic actions and nested actions somewhat based on the model defined by Moss in his thesis [Mo81]. *Clouds* v.1. extended Moss's model by allowing custom tailored synchronization and recovery, as well as interactions between actions and non-actions.

The synchronization and recovery properties can be localized in objects, on a per object basis. The synchronization and recovery can be handled by the system (to adhere to Moss's semantics) or can be tailored by the user and thus provide facilities beyond those allowed by standard nested transactions. Customization is allowed by labeling of objects as "auto-sync" or "custom-sync" and "auto-recoverable" and "custom-recoverable". Further details can be found in [Wi87].

5.2. Atomicity in *Clouds* v.2.

The support for atomicity in *Clouds* v.2. has its roots in the above scheme, but has been changed in some respects. The following is a brief outline of the scheme. The actual methods used are discussed in greater detail in [ChDa87].

Instead of mandating customization of synchronization and recovery for application that cannot use strict atomicity semantics, the new scheme support a variety of *consistency preserving* mechanisms. The threads that execute are of two kinds, namely *s-thread* (or *standard threads*) and *cp-threads* (or consistency-preserving threads). The *s-threads* have a "best effort" execution scheme and are not provided with any system-level locking or recovery. The *cp-threads* on the other hand are supported by locking and recovery schemes, provided by the system. When a *cp-thread* executes, all pages it reads are read-locked and the pages it updates are write-locked. The updated pages are written using a 2-phase commit mechanism when the *cp-thread* completes.

The data in the system has an *instantaneous* version and a *stable* version. In fact, if nested threads are used, the data has a stack of versions, the top being the instantaneous version and the bottom being the stable version. All the threads work on the instantaneous version. The data updated by *cp-threads* are committed when the *cp-thread* exits, while the data touched by the *s-threads* are committed "eventually", using a best effort semantics.

The *cp-threads* are allowed to interleave with *s-threads*, and also the *cp-threads* can be used to provide heavyweight as well as lightweight atomicity, using *gcp* and *lcp* operations, described below.

All threads are *s-threads* when created. The handling of *cp-threads* are programmed by the following scheme. All operations in objects in *Clouds* are tagged with a consistency label, the labels used are:

- Globally-Consistent (gcp)
- Locally-Consistent (lcp)
- Standard (s)
- Inherited (i)

An object can have any number of different labels on the operations. Also the same operation may have multiple entry points, labeled at different atomicity levels.

A s-thread executing a gcp or lcp operation converts to a cp-thread. A thread entering a lcp entry point, commits its updates (inside this object) as soon as it exits the object. This provides intra-object consistency rather than the inter-object consistency provided by the gcp operations, and thus is a cheap method of updating one object atomically. Locking and recovery are automatic.

The standard entry points do not support any locking or recovery. They can make use of "best-effort" semantics. They can also be used for non-traditional purposes such as peeking at incomplete results of actions (as they are not hindered by locking and visibility rules of actions). Locks are available for synchronizing non-actions, but recovery is not supported.

The other labels as well as combination of these labels in the same object (or in the same thread) lead to many interesting (as well as dangerous) variations. The complete discussion of the semantics as well as the implementation is beyond the scope of this paper, and the reader is referred to [ChDa87]

6. Programming Support

Systems and application programming for *Clouds* involves programming objects that implement the desired functionality. These objects can be expressed in any programming language. The compiler (or the linker) for the language, however, must be modified to generate the stubs for the various entry points, invocation handler, system call interfaces and the inclusion of default systems function handling code (such as synchronization and recovery.)

The language Aeolus has been designed to integrate the full set of powerful features that the *Clouds* kernel supports. Aeolus currently supports the features of *Clouds* v.1. but is being expanded for added functionality of Ra and *Clouds* v.2. [LeWi85, Wi85, WiLe86].

Aeolus is the first generation language for *Clouds*. It does not support some of the features found in object-oriented programming systems such as inheritance and subclassing. Providing support for these features at the language level is currently under consideration.

7. Implementation Notes

The implementation of the *Clouds* operating systems has been based on the following guidelines:

- The implementation of the system should be suitable for general purpose computers, connected through popular networking hardware. Heterogeneous machines, though not crucial, should be allowed.
- Since the *Clouds* functionality is largely based on object invocation, support for objects should be efficient in order to make the system usable. Also, the naming, synchronization and recovery systems should be implementable with minimal overhead.

- Since one of the primary aims of *Clouds* is to provide the substrate for reliable, fault tolerant computing, the kernel and the operating system should provide adequate support for implementing fault tolerance. (Fault tolerance is not discussed in this paper, the reader is referred to [AhDa87].)
- The system design should be simple to comprehend and implement.

7.1. Hardware Configuration

Clouds v.1. was built on a three VAX-11/750 computers, connected through an Ethernet, equipped with RL02 and RA81 disk drives. The user interface was through the Ethernet, accessible from any Unix machine.

Clouds v.2. will be implemented on a set of Sun-3 class machines. The cluster of *Clouds* machines will be on an Ethernet, and user will be able access them through workstations running *Clouds* as well as any Unix workstation.

7.2. Software Configuration and Kernel Structure

The kernel (version 2.) used to support *Clouds* is called *Ra*. *Ra* is a native kernel running on bare hardware. The kernel is implemented in C for portability, and because the availability of C source for the UNIX kernel simplified the task of developing hardware interfaces such as device drivers.

The kernel runs on the native machine and not on top of any conventional operating system for two reasons. Firstly, this approach is efficient. As *Clouds* does not use much of the functionality of conventional operating systems (such as file systems), building *Clouds* on top of a Unix-like kernel make poor use of the host operating system. Secondly, the paradigms and the support for synchronization, recovery, shared memory and so on; used in *Clouds* are considerably different from the functionality provided by conventional operating systems, and major changes would be necessary at the kernel level of any operating system in order to implement *Clouds*.

The Ra kernel provides support for partitions, segments, virtual spaces, processes and threads. These are the basic building blocks for *Clouds*. The partitions provide non-volatile storage, the segments provide memory storage, which are used to build objects, which in turn reside in virtual spaces. Processes provide activity which are used to compose threads. A description of the design of Ra can be found in [BeHuKh87]

7.3. Object Naming and Invocation

The two basic activities inside the Ra kernel are system call handling and object invocations. System call handling is done locally, as in any operating system. The system calls supported by the Ra kernel include object invocation, memory allocation, process control and synchronization, and other localized systems functions. Object invocation is a service provided by the kernel for user threads. The attributes that object invocation satisfy are:

- Location independence.
- Fast, for both local and remote invocations.
- Failed machines should not hamper availability of objects on working sites, from working sites.
- Moving objects between sites, reassigning disk units and so on should be simple (for reconfiguration and fault tolerance support).

Location independence is achieved through a capability based naming system. Availability is obtained through decentralization of directory information and a search-and-invoke strategy coupled with a multicast based object location scheme, designed for efficiency [AhAm87]. Speed is achieved by implementing the invocation handlers at the lowest level of the kernel, on the native machine.

7.4. Storage Management

The storage management system handles the function required to provide the reliable, permanent object address spaces. As mentioned earlier, unlike conventional systems, where virtual address spaces are volatile and short-lived, *Clouds* virtual spaces contain objects and are permanent and long lived. The first version of the implementation is detailed in [Pi86].

The storage management system stores the object representations on disk, as an image of the object space. When an object is invoked, the object is demand paged into its virtual space as and when necessary. As the invocation updates the object, the updated pages do not replace the original copy, but have shadow copies on the disk. The permanent copy is updated only when a commit operation is performed on the object. The storage manager provides the support to commit an object using the two-phase commit protocol.

7.5. User Interfaces

User interfaces can make or break an operating system. Users do not like to switch systems, and have to re-learn the interfaces. We plan to use Unix and X-windows as our interface to *Clouds*. Unix programs can make use of *Clouds* facilities through invocation support provided by a *Clouds* library on Unix. Also, *Clouds* utilities will be available under X-windows. This will have several implications:

Firstly, *Clouds* can be treated as a back-end system to the Unix workstation, for distributed processing, computations, object-oriented programs and atomic programs. All these facilities will be available to Unix programs and the user.

Second, the user can access *Clouds* utilities through the X-window system, and thus making the learning time much smaller. We believe this approach will make *Clouds* easier to access and use, and we hope to build a large user community that is essential to the success of new operating systems.

8. Comparisons with Related Systems

Clouds is one of the several research projects that are building object-based distributed environments. Although there are differences between all the approaches, we feel that the area of distributed operating systems is not mature enough to conclusively argue the superiority of one approach over the other. In the following paragraphs we document the major differences between *Clouds* and some of the better known projects in distributed systems. (This list is not exhaustive).

One of the major difference between *Clouds* and some of the systems mentioned below is in the implementation of the kernel. Many systems implement the kernel as a Unix process[†], while *Clouds* is implemented as a native operating system (as are Mach and Alpha). *Clouds* is not intended to be an enhancement, or replacement of, the UNIX kernel. Instead, *Clouds* provides a different paradigm from that supported by UNIX (e.g., the UNIX paradigms of 'devices as files', unstructured files, volatile address spaces, pipes, redirection etc.)

8.1. Argus

Argus is a language for describing objects, actions and processes using the concept of a guardian. The language defines a distributed system to be a set of guardians, each containing a set of handlers. Guardians are logical sites, and each guardian is located at one site, though a site may contain several guardians. The handlers are operations that can access data stored in the guardian. The data types in Argus can be defined to be atomic, and atomic data types changed by actions are updated atomically when the action terminates [WeLi83, LiSc83]. The support for Argus is built on top of Unix, and provides all the facilities of the Argus language [Li87].

Some of the similarities between Argus and *Clouds* are in the semantics of nested actions. Both use the nested action semantics and locking semantics that are derived from Moss. This includes conditional commit and lock inheritance. However the consistency preserving mechanisms in *Clouds* have moved away from Moss's action semantics, substantially, though retaining the nested action semantics as a subset. Also the guardians and handlers in Argus have somewhat more than cosmetic similarities to objects in *Clouds*, as the design of *Clouds* was influenced by Argus.

The differences include the implementation strategies, programming support and support for reliability. The scheme of permanent virtual spaces provided by passive objects is a major difference. As mentioned earlier, Argus is implemented on top of a modified Unix environment. This is one of the reasons for the somewhat marginal performance of the Argus system observed in [GrSeWe86]. The programming support provided by Argus is for the Argus language. *Clouds* on the other hand is a general purpose operating system, not tied to any language. Though Acolus is the preferred language at present, we have used C extensively for object programming. We have plans to implement more object-oriented languages for the the *Clouds* system.

8.2. Eden

Eden is a object-based distributed system, implemented on the Unix operating system at the University of Washington. Eden objects (called Ejects) use the active object paradigm, that is each object consists of a process and an address space. An invocation of the object consists of sending a message to the (server) process in the object, which executes the requested routine, and returns the results in a reply [Alm83, AIB183, NoPr85].

Since every object in the system needs to have a process servicing it, this could lead to too many processes. Thus Eden has an *active* and a *passive* representation of objects. The passive representation is the core image of the object stored on the disk. When an object is invoked, it must be active, thus invoking a passive object involves activating it. A process is created by 'exec'-ing the core image of the object (frozen earlier), and then performs the required operation. The activation of passive objects is an expensive operation. Also concurrent invocations of objects are difficult and are handled through multithreaded processes or coroutines.

[†]The term *kernel* has been used quite frequently to describe the core service center of a system. However when this service is provided by a Unix process rather than a resident, interrupt driven monitor, the usage of the term is somewhat counter-intuitive.

The active object paradigm and the Unix-based implementation are some of the major differences between Eden and *Clouds*. Eden also provides support for transaction and replication objects (called *Replicets*). The transaction support and replication were added after the basic Eden system was designed and have some limitations due to manner Unix handles disk I/O.

8.3. Cronus

Cronus is an operating system designed and implemented at BBN Laboratories. Some of the salient points of Cronus are the integration of Cronus functions with Unix functions, the ability of Cronus to handle a wide variety of hardware and the coexistence of Cronus on a distributed set of machines running Unix, as well as several other host operating systems [BeRe85, GuD:86, SeTh86].

Like Eden, Cronus uses the active objects. This is necessary to be able to make Cronus run on top of most host operating systems. Cronus objects are handled by managers. Often a single manager can handle several objects, by mapping the objects into its address space. The managers are servers and receive invocation requests through catalogued ports. Any Unix process on any machine on the network can avail of Cronus services from any manager, by sending a message to the appropriate manager. By use of canonical data forms, the machine dependencies of data representations are made transparent. Irrespective of the machine types, any Unix machine can invoke Cronus objects in a location independent fashion.

8.4. ISIS

ISIS (version 1) is a distributed operating system, developed at Cornell University, to support fault tolerant computing. ISIS has been implemented on top of Unix. It uses replication and checkpointing to achieve failure resilience. If data object is declared to be *k*-resilient, the system creates *k*+1 copies of the object. The replicated object invocation is handled by invoking one replica and transmitting the state updates to all replicas. Checkpointing at each invocation is used to recover from failures [Bi85].

The goals and attributes of ISIS are different from *Clouds*. ISIS is built on top of some interesting communication primitives and is not built as a general purpose computing environment.

8.5. ArchOS and Alpha

Alpha is the kernel for the ArchOS operating system developed by the Archons project at Carnegie Mellon University. Like *Clouds*, the Alpha kernel is a native operating system kernel designed to run on the special hardware called Alpha-nodes. The Alpha kernel uses passive objects residing in their own virtual spaces, similar to *Clouds*. ArchOS is designed for real time applications supporting specialized defense related systems and applications [Je85, No87].

The key design criteria for ArchOS and Alpha are time critical computations and rather than reliability. Fault tolerance is handled to an extent using communication protocols. Real time scheduling has been a major research topic at the Archons project.

8.6. V-System

The V operating system has been developed at Stanford University. V is a compromise between message-based systems and object-based systems. The basic core of V provides lightweight processes and a fast communications (message) system. V message semantics are similar to object invocations in the sense that the messages are synchronous and use the send/reply paradigm. The relationship between processes conforms to the client-server paradigm. A client sends a request to the server, and the client blocks until the server replies [Ch88].

V allows multiple processes to reside in the same address space. Data sharing is through message passing, though shared memory can be implemented through servers managing bounded buffers. The design goals of V are primarily speed and simplicity. V does not provide transaction and replication support. These can be implemented, if necessary at the application level.

The radical difference between V and *Clouds* is the paradigm used by *Clouds*.

8.7. Mach

Mach is a distributed operating system under development at Carnegie Mellon [Ac86]. Mach maintains object-code compatibility with Unix. Mach extends the Unix paradigms by adding large sparse address spaces, memory mapped files, user provided backing stores, and memory sharing between tasks. Mach is implemented on a host of processors including multiprocessors.

The execution environment for a Mach activity is a task. Threads are computation units that run in a task. A single thread in a task is similar to a Unix process. Ports are communication channels, supporting messages which are typed collection of data objects. In addition, Mach supports memory objects, which are collections of data objects managed by a server.

Support for transactions are not built into Mach, but can be layered on top of Mach and has been implemented by Camelot and Avalon [HeW:87].

The approaches used by Mach and *Clouds* are fundamentally different, as with V and *Clouds*.

9. Concluding Remarks

Clouds provides an environment for research in distributed applications. By focusing on support for advanced programming paradigms, and decentralized, yet integrated, control, *Clouds* offers more than 'yet another Unix extension/look-alike'. By providing mechanisms, rather than policies, for advanced programming paradigms, *Clouds* provides systems researchers an adaptable, high-performance, 'workbench' for experimentation in areas such as distributed databases, distributed computation, and network applications. By adopting 'off the shelf' hardware, the portability and robustness of *Clouds* are enhanced. By providing a 'Unix gateway', users can make use of established tools. The gateway also relieves *Clouds* from the necessity of providing emulating services such as provided by Unix mail and text processing.

The goal of *Clouds* has been to build a general purpose distributed computing environment, suitable for a wide variety of user communities, both within and outside the computer science community. We are striving to achieve this through a simple model of a distributed environment with facilities that most users

would feel comfortable with. Also we are planning to experiment with increased usage of the system by making it available to graduate courses, and hope the feedback and the criticism we receive from a large set of users will allow us to tailor, enhance and perhaps redesign the system to fit the needs for distributed computing, and thus give rise to wider usage of distributed systems.

10. Acknowledgements

The authors would like to acknowledge Martin McKendry and Jim Allchin for starting the project and designing the first version of Clouds. Gene Spafford and Dave Pitts for the implementation, Jose Bernabeau, Yousef Khalidi and Phil Hutto for their efforts in making the kernel usable and for the design of *Ra*. Also Mustaque Ahmad, Ray Chen, Kishore Ramachandran and Henry Strickland for their participation in the project.

11. References

- [Ac86] Accetta M, et. al. *Mach: A New Kernel Foundation for Unix Development*, Technical Report, Carnegie Mellon University.
- [AhAm87] M. Ahmad, M. Ammar, J. Bernabeu and M. Y. Khalidi, *A Multicast Scheme for Locating Objects in a Distributed System*. Technical Report GIT-ICS-87/01, School of Information and Computer Science, Georgia Tech, January 1987.
- [AhDa87] M. Ahmad and P. Dasgupta, *Parallel Execution Threads: An Approach to Atomic Actions*. Technical Report GIT-ICSD-87/16. School of Information and Computer Science, Georgia Tech.
- [Alm83] G. T. Almes, *The Evolution of the Eden Invocation Mechanism*, Technical Report 83-01-03, Department of Computer Science, University of Washington, 1983.
- [Al83] J. E. Allchin, *An Architecture for Reliable Decentralized Systems*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, (Also released as technical report GIT-ICS-83/23,) 1983.
- [AlBl83] G. T. Almes, A. P. Black and E. D. Lazowska and J. D. Noe, *The Eden System: A Technical Review*, University of Washington Department of Computer Science, Technical Report 83- 10-05 October 1983.
- [BeHKh87] J. M. Bernabeau Auban, P. W. Hutto and M. Y. A. Khalidi, *The Architecture of the Ra Kernel*, Technical Report GIT-ICS-87/35 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.
- [BeRe85] J. C. Berets, R. A. Mucci and R. E. Schantz, *Cronus: A Testbed for Developing Distributed Systems*, October 1985 IEEE Communications Society, IEEE Military Communications Conference.
- [Bi85] K. P. Birman and others, *An Overview of the ISIS Project*, Distributed Processing Technical Committee Newsletter, IEEE Computer Society (7,2) October 1985 (Special issue on Reliable Distributed Systems).
- [Ch88] D. Cheriton *The V Distributed System*. Communications of the ACM, March 1988.
- [ChDa87] R. Chen and P. Dasgupta, *Consistency-Preserving Threads: Yet Another Approach to Atomic Programming*, Technical Report GIT-ICS-87/43 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.
- [DLS86] P. Dasgupta, *A Probe-Based Fault Tolerant Scheme for an Object-Based Operating System*, Proceeding of the 1st ACM Conference on Object Oriented Programming Systems, Languages and Applications. Portland OR. 1986.
- [GuDe86] R. F. Gurwitz, M. A. Dean and R. E. Schantz, *Programming Support in the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.
- [GrSeWe86] I. Greif, R. Seliger and W. Wehl *Atomic Data Abstractions in a Distributed Collaborative Editing System*, (Extended Abstract) Conference Record of the Thirteenth Symposium on Principles of Programming Languages, ACM SIGACT/SIGPLAN, January 1986, St. Petersburg Beach, FL.
- [HeWi87] M. P. Herlihy and J. M. Wing, *Avolon: Language Support for Reliable Distributed Systems*. Proceedings of the 17th International Symposium on Fault-Tolerant Computing, July 1987.
- [Je85] E. D. Jensen et. al. *Decentralized System Control*, Technical Report RADC-TR-85-199, Carnegie Mellon University and Rome Air Development Center, April 1985.
- [LeWi85] R. J. LeBlanc and C. T. Wilkes, *Systems Programming with Objects and Actions*, Proceedings of the Fifth International Conference on Distributed Computing Systems, Denver, July 1985. (Also released, in expanded form, as technical report GIT-ICS-85/03)
- [LiSc83] B. Liskov and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust Distributed Programs*, ACM, Transactions on Programming Languages and Systems (53) July 1983.
- [Li87] B. Liskov, D. Curtis, P. Johnson and R. Scheifer, *Implementation of Argus*. Proceedings of the 11th ACM Symposium on Operating Systems Principles. November 1987.
- [Mc84] M. S. McKendry, *Clouds: A Fault-Tolerant Distributed Operating System*, Distributed Processing Technical Committee Newsletter, IEEE, 1984, (Also issued as Clouds Technical Memo No:42).
- [Mo81] J. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.
- [NoPr85] J. D. Noe, A. B. Proudfoot and C. Pu, *Replication in Distributed Systems: The Eden Experience*, Department of Computer Science, University of Washington, Seattle, WA, September 1985 Technical Report TR-85-08-06.
- [No87] Northcutt J. D. *Mechanisms for Reliable Distributed Real-Time Operating Systems - The Alpha Kernel*, Perspectives in Computing, v16. Academic Press, 1987.
- [Pi86] D. V. Pitts, *Storage Management for a Reliable Decentralized Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986, (Also released as Technical Report GIT-ICS-86/21).
- [RaKh88] U. Ramachandran and Y. A. Khalidi, *Memory Management Support for Object Invocation*. Technical Report GIT-ICS-88/03. School of Information and Computer

Science, Georgia Tech.

- [ScTh86] R. E. Schantz, R. H. Thomas and G. Bono, *The Architecture of the Cronus Distributed Operating System*, May 1986, Proceedings of the Sixth International Conference on Distributed Computer Systems, IEEE Computer Society.
- [Sp86] E. H. Spafford, *Kernel Structures for a Distributed Operating System*, Ph.D. Diss., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. (Also released as technical report GIT-ICS-86/16).
- [WeLi83] W. Weihl and B. Liskov, *Specification and Implementation of Resilient Atomic Data Types*, Symposium on Programming Language Issues in Software Systems, June 1983.
- [Wi85] C. T. Wilkes, *Preliminary Aeolus Reference Manual*, Technical Report GIT-ICS-85/07, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. (Last Revision: 17 March 1986)
- [WiLe86] C. T. Wilkes and R. J. LeBlanc, *Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System*, Proceedings of the IEEE Computer Society 1986 International Conference on Computer Languages. (Also available as Technical Report GIT-ICS-86/12, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.)
- [Wi87] C.T. Wilkes *Programming Methodologies for Resilience and Availability*. Ph.D.thesis, Georgia Tech, 1987, Technical Report GIT-ICS-87/32 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA.
- [Wu74] W. A. Wulf and others, *HYDRA: The Kernel of a Multiprocessor Operating System*, Communications of the ACM, (17,6) June 1974.
- [WuLe81] W. A. Wulf, R. Levin and S. P. Harbison, *HYDRA/C.mmp, An Experimental Computer System*, McGraw-Hill, Inc., 1981.

To appear in ACM TODS

The *Five Color* Concurrency Control Protocol: Non-Two-Phase Locking in General Databases.

Partha Dasgupta¹ & Zvi M. Kedem²

Abstract

Concurrency control protocols based on two-phase locking are a popular family of locking protocols that preserve serializability in general (unstructured) database systems. This paper presents a concurrency control algorithm (for databases with no inherent structure) that is practical, non-two-phase and allows varieties of serializable logs not possible with any commonly known locking schemes. All transactions are required to predeclare the data it intends to read or write. Using this information, the protocol anticipates the existence (or absence) of possible conflicts, and hence can allow non-two-phase locking.

It is well known that serializability is characterized by acyclicity of the serializability graph representation of interleaved executions. The two-phase locking protocols allow only *forward* growth of the paths in the graph. The *Five Color* protocol allows the serializability graph to grow in any direction (avoiding two-phase constraints) and prevents cycles in the graph by maintaining transaction access information in the form of data-item markers. The read and write set information can also be used to provide relative immunity from deadlocks.

Categories and Subject Descriptors: H.2.4[Database Management]: Systems -- concurrency, transaction processing.

General Terms: Concurrency Control.

Additional Key Words and Phrases: serializability, locking.

This research was partially supported by NSF under grants MCS 81-04882, MCS 81-10097, CCR-89 ONR under contract number N0014-85-K-0046, and NASA under contract NAG-1-430.

1: School of Information and Computer Science

Georgia Institute of Technology

Atlanta, GA 30332

2: Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

New York, NY 10012

1. Introduction

In this paper we present a locking protocol for database concurrency control, that applies to general databases. The locking strategy is non-two-phase. The protocol uses five kinds of locks. The five categories of locks are read-locks, write-locks, intent-locks and two types of marker locks. We use five colors to assign mnemonic names to these locks.

The protocol requires each transaction to predeclare the data it intends to read or write. This can be achieved by data analysis by the query compiler. The predeclared readset and writerset need not be the exact read/write sets, but can be a superset of the actual sets. The performance however depends upon the closeness of the predeclared sets and the actual sets.

Unlike the two-phase locking protocol, the *Five Color* protocol uses early release of read locks and late acquisition of write locks to enhance concurrency. The early release of read locks makes this protocol violate the two-phase locking rule. This feature however has to be closely controlled, as it can cause non-serializable behavior. It is widely known that two-phase locking "is in a sense, the best that can be formulated" (Ullman in [Ul82] pg. 380). The optimality of two-phase locking implies that in the absence of any information about the transactions or the database, all locking protocols must be two-phase [Ya82]. The Five Color protocol allows non-two-phase locking by keeping track of transaction ordering using the predeclared read and write sets; and by addition of a check called validation. We first present a brief introduction to the concepts of serializability, the model of a multiuser database, the factors that limit concurrency in two-phase locking and some related work. Section 2 contains a comprehensive description of the *Five Color* protocol, including an intuitive description of how it functions and why it ensures serializability. Section 3 explains the formal properties of the protocol and derives a proof of correctness and section 4 outlines a modification to the protocol. Sections 5 and 6 deal with deadlocks and livelocks and Section 7 discusses performance issues.

1.1. Serializability

A database is viewed as a collection of data items, which can be read or written by concurrent transactions. Interleaving of updates can leave the database in an inconsistent state. A sufficient condition to guarantee correctness of concurrent database access is *serializability* of the actions (reads or writes) performed by the transactions on the data items. That is, the interleaved execution of the transactions should be equivalent to some serial execution of the transactions [BeGo80, RoStLe78]. In this paper, we assume serializability to be the criterion of correctness.

Locking of data items is one of the methods of achieving consistency in the face of concurrent updates. For databases with no inherent structure (e.g. databases not organized as DAG's, trees, etc.) the two-phase locking protocol is the most popular locking protocol. However, two-phase locking is restrictive with respect to the amount of concurrency it allows.

Informally, a *log* is a sequence of actions issued by various transactions on several data items in the database. The transaction actions may be interleaved with one another. Serializability is a syntactic property of a log. It has been shown that recognizing serializability is an NP-complete problem [PaBeRo77, Pa79]. The NP-completeness of the serializability recognition problem implies that we cannot have a scheduler that allows all serializable logs and disallows non-serializable ones, and works in polynomial time (unless $P=NP$). However certain subclasses of serializable logs are efficiently recognizable in polynomial time. Efficient algorithms can be built that control the actions of transactions, to ensure that the logs produced by a set of transactions fall into one of these easily recognizable classes of serializability. The two-phase locking protocol is one such algorithm which produces a class of polynomially recognizable serializable logs, namely the two-phase locked logs.

1.2. The Model

A *database* D is a set of distinct data items $\{x_1, x_2, \dots, x_m\}$. A *transaction system* T is a set of transactions $\{T_1, T_2, \dots, T_n\}$ that operate on the database. The *readset* (*writeset*) of a transaction T_i is the set of all items T_i reads (writes).

A transaction that intends to read (or write) a data item x , issues a read (or write) request to the *transaction manager*. The transaction manager is responsible for determining whether or not granting of the request may cause a violation of the correctness criterion (generally serializability). The transaction manager then takes appropriate action by granting, rejecting or delaying the request.

A *trace* of a transaction is a sequence of successful read and write requests it makes to the transaction manager. A trace is written as a sequence of actions of the form $R_i(x)$ or $W_i(x)$, where $R_i(x)$ (or $W_i(x)$) means a transaction T_i issues a read (write) on data item x . Note that we are not interested in the values read or written, but in the syntactic properties of the string of reads and writes on the data items.

We will assume at most one read and at most one write per data item in any trace. If a transaction reads as well as writes a particular data item, we assume the read will precede the write. Multiple reads and writes are handled in an obvious way: The first read is used to read the value of the data item and store it in local storage, and the other reads on the same data item are processed locally. Similarly, all writes except the last one are written to local storage, and the last one appears on the log. Thus there is no loss of generality.

A *log* of a transaction manager is a sequence of reads and writes granted by the transaction manager. As an example, three transaction traces and one possible transaction manager log are depicted below: (the notation is from Bernstein *et al.* [BeGo80].)

$T_1 : R_1(x) R_1(y) W_1(y)$
 $T_2 : R_2(y) W_2(y)$

$T_3 : R_3(x) R_3(y) R_3(z) W_3(x)$

Log : $R_1(x) R_2(y) W_2(y) R_3(x) R_1(y) R_3(y) W_1(y) R_3(z) W_3(x)$

Note that when $A_n(x)$ precedes $A_n'(x')$ in some transaction trace then $A_n(x)$ has to precede $A_n'(x')$ in any log in which the transaction participates.

1.3. Increasing Concurrency

In database concurrency control we are interested in protocols that maximize concurrency and work efficiently. All known concurrency control protocols restrict the logs to a subset of the all possible serializable logs. Two phase locking has been deemed to be quite restrictive since, intuitively, it holds locks for "longer than necessary". That is, after a data item is read, the lock on it is held till no other locks will be necessary (typically until commit). Indeed this is necessary for providing serializability if no other mechanisms other than locking the data items accessed are being used to control concurrency. In addition, a two-phase locking protocol can cause deadlocks, which further degrade its performance.

We contend that this seemingly restrictive nature of two-phase-locking arises from the fact that the protocol does not assume any *a priori* knowledge of the intentions of the transactions. *A priori* knowledge of read and write sets can be used deadlock prevention in two phase protocols. We use *a priori* knowledge to allow non-two-phase executions, in particular:

- Allowing read locks to be released as soon as the data is read.
- Allowing reading of (some) write locked data items.
- Preventing deadlocks due to lock acquisition (another form of deadlock is possible, see section 5.)

The following (trivial) example illustrates the restrictive nature of two-phase locking, and shows how added information can be used to remove some of the restrictions:

$T_1 : R_1(x), W_1(x), \dots, \dots, R_1(y), W_1(y)$
 $T_2 : R_2(x)$
 $T_3 : R_2(y)$

In this case, T_2 (and T_3) reads the value of x (and y), and does nothing else. Thus T_2 and T_3 can read x (or y) between any two actions of T_1 , and still produce a correct serializable execution sequence. However if two-phase locking is used, then T_2 (or T_3) is restricted to read x (or y) at only certain points, depending upon the locking sequence. For example suppose T_1 uses the locking sequence shown in Figure 1. (LS(x) denotes setting a shared lock on x , LX(x) denotes setting an exclusive lock on x , and U(x) denotes unlocking of x .)

Data-x	Data-y	Comments
LS ₁ (x)		
R ₁ (x)		
LX ₁ (x)		
W ₁ (x)		↑
	LS ₁ (y)	
	R ₁ (y)	T ₂ cannot read x here.
	LX ₁ (y)	↓
U ₁ (x)		
	W ₁ (y)	
	U ₁ (y)	

Figure 1: A locking sequence used by transaction T₁.

This locking sequence prevents T₂ from reading x, between W₁(x) and R₁(y). However T₁'s locking sequence can be changed so that T₂ can read x between W₁(x) and R₁(y), but then T₃ will not be allowed to read y between W₁(x) and R₁(y).

Since we *know* that T₂ and T₃ read x and y and do nothing else, we could allow T₂ and T₃ to read x and y interleaved between any steps of T₁ [GaWi82]. The situation would be different if T₂ or T₃ accessed or updated some other data items after reading x or y. However the two-phase locking protocol does not rely upon, or have access to, information about the complete data access patterns of the transactions. We will show how data access information can be used by concurrency control protocols to deal with situations like the one shown above.

The above example also shows that locking sequences of two-phase locked transactions affect the amount of concurrency, depending on the transaction mix. A particular locking sequence in fact may favor one transaction over another. Since it is impossible to predict which transactions will run concurrently, it is not easy to choose locking sequences. In fact, it is commonly believed that, locking should not be handled by transactions or application programs, but should be the responsibility of lower level, consistency preserving routines i.e. the transaction manager.

In order to make locking transparent, practical two-phase locking schemes use read and write requests to obtain locks [see 2V2PL section 1.4]. A read or write request on an

unlocked data item causes the lock to be obtained. All locks are released only when the transaction terminates. Thus it is intuitively clear that the locks are held for extended periods.

In the following sections, we propose a concurrency control protocol that, in cases like the above, would allow T_2 and T_3 to read x and y interleaved between any action of T_1 . This would be achieved by either early release of read locks, or by allowing reading of (older values) of write locked data. The locking is handled entirely by the transaction manager. Holding of read locks on read only items is minimized. The protocol is inherently non-two-phase, and is relatively immune from deadlocks.

1.4. Related Work

Database concurrency control has been an active area of research, and has resulted in the development of many protocols for achieving serializability. These basic mechanisms used by the protocols are locking, timestamps and multiple versions [BeGo80, BeGo82, BeHaGo87, UI82]. A synopsis of the concurrency control methods is beyond the scope of this paper.

A family of protocols called two-version protocols use "before" and "after" values of the data for concurrency control. The Five Color protocol has a similarity with some of them. Also, some protocols use information about the transaction or the database to enhance concurrency. The Five Color protocol uses information about the transactions (in the form of predeclared read and write sets). The following paragraphs contain brief descriptions of protocols in these two broad families.

Bayer, Heller and Reiser [BaHeRe80] presents a protocol where the reads from a transaction are never delayed, and are granted immediately. They use three kinds of locks namely Read, Analyze and Commit locks. The Read locks are compatible with all the other locks, thus readers can see the before value of any transaction that is writing to a data item and has an Analyze lock. Readers can read while the writer is committing, and a graph is maintained to assure they either read all committed versions or all before versions.

A protocol similar to the Bayer *et. al.* protocol is described in [BeHaGo87]. This protocol is called the two-version two-phase locking protocol (2V2PL). Under this protocol, the transaction manager uses read and write requests from the transactions to set locks. A read request causes a shared read lock to be obtained. A write request causes a write lock to be obtained. (If the transaction already possesses a read lock in the item, the lock is upgraded to a write lock.) The write lock is compatible with other read locks, but not with a write lock. Reader's reading the write locked item, gets to see the older (before) value. After the transaction commits, the write locks are upgraded to an exclusive "Certify" lock which is incompatible with other locks. The incompatibility of the Certify lock does away with the need for maintaining a dependency graph, and introduces a slight delay for transactions that want to read an item that is in the process of being committed.

In both the protocols, the locking is two-phase; that is, the readers hold read-locks until they commit and writers cannot commit until all the read locks on the items they are updating are released. This causes writing transactions to wait for the readers to terminate. This can also cause writer starvation when the read traffic is high. The Five Color protocol is designed to avoid this situation, as all the read-locks are released as soon as the reads are over, and writers do not have to wait to commit. Also upgrading of locks lead to "trivial" deadlocks [GrHoKoOb81, Ko83]. That is if two transactions hold read locks on a data item and both try to upgrade it to write, then the situation is a deadlock. The Five Color Protocol allows lock upgrading, but avoids trivial deadlocks. Of course these features are possible only because of the read-write set predeclaration.

Stearns and Rosenkrantz [StRo81] describe a set of actions and conditions that can be used to control concurrency when using before-values. For each conflict they define a set of "actions" that can be taken (non-deterministically). They also define a set of conditions for proper consistency. Some of the actions may cause an abort or rollback at a later stage. These sets of rules gives rise to a family of protocols, and the performance is dependent upon the choice of actions taken in the particular execution. The concurrency control can be thought of as a non-deterministic table driven mechanism. The tables show the actions for each case of conflict. The cases of conflict are: a younger transaction reading a before value of an older transaction; an older transaction reading the before value of a younger transaction and so on. The actions are to allow the read or write, abort one of the transaction or delay the requester. Some of the actions have restrictions as to the "phase" of the requester, and in some cases one of many actions can be chosen. Timestamps are used to decide the older-younger relationships.

It has been shown that some available information about the transactions or the database can be used for increasing concurrency. For instance Kedem and Silberschatz have developed the tree and DAG (directed acyclic graph) protocols that can be used on databases structured like a tree or a DAG, respectively [SiKe80, KeSi82]. These protocols allow non-two-phase locking, and provide higher concurrency than two-phase locking for transactions that traverse the tree or the DAG. In the DAG protocol a transaction is allowed to lock a child if it has locks on the majority of its parents (except for the first node locked), and unlocking may be done in any order. Thus the transaction can access only those data items that form a rooted subgraph of the original graph. These and similar protocols can be used in specialized applications and have practical limitations, but have received substantial theoretical interest.

If the writeset of a transaction is known in advance, timestamp protocols can be made abort free (or *progressive*) [BuSi83]. This can provide significant improvement in performance as aborts cause severe limitations of throughput in timestamp based systems.

Semantic knowledge about transaction actions has also been used to speed up transaction processing by Garcia-Molina [Ga83]. A partial loss of serializability can be tolerated in some applications and can be used for better performance [FiMi82]. This may not be of interest in

general purpose databases, where consistency is a major issue.

Read and write sets are used to control concurrency in SDD-1 (A System for Distributed Databases) [BeShRo80]. Transactions are divided into classes depending upon their read and write sets and then conflict graph analysis is performed. This is a *static* classification and is used to determine the nature of the conflicts. The conflict type is used to determine which protocol to use. (SDD-1 uses different protocols for different situations.) SDD-1 is a timestamp based system. In our approach the usage of read and write set information is dynamic. The protocol uses locking and static analysis of conflicting transactions is not performed.

2. The *Five Color* Protocol

The *Five Color* Protocol is a non-two-phase locked protocol that ensures serializability in general (unstructured) databases. It derives its name from the five types of locks it uses.

A transaction T acts upon a set of data items D . A data item $x \in D$ is in the readset (Rd) of a transaction T (that is $x \in Rd(T)$) if the transaction intends to read x . Any item that the transaction T intends to write is contained in the writeset (Wr) of T . Note that $Rd(T)$ need not be a subset of $Wr(T)$ or *vice versa*, and $Rd(T)$ and $Wr(T)$ may be supersets of the data items actually read and written by the transaction T .

Each transaction is required to declare its readset and writeset to the transaction manager before it issues any actions. The read and write sets can be parameters to the *begin-transaction* statement that is executed by a transaction when it starts. Since the readset (and writeset) are allowed to be supersets of the data items actually read (and written) by the transaction, they could be statically determined during query compilation. Sometimes the transaction may read and write different sets of data depending upon some statically undeterminable conditions. In this case the declared read and write sets should include all the items the transaction may act upon. However, for better performance the difference between the declared and actual read and write sets should be small. Also, the *Five Color* Protocol transaction manager reads *all* all the items in the readset of a transaction, and thus will cause extra reads if the readset is larger than necessary.

2.1. The Basic Algorithm

After the transaction manager knows the read and write sets of the transaction, it can obtain shared locks on all the data items in the readset, read them and store the values in local storage. Then we would like to release the shared locks, as the locks *seem* no longer necessary. Also, we would also like to keep the data items in the writeset locked by a shared (intend-to-write) lock while the transaction is running, to prevent other concurrent transactions from updating them and causing missing updates. The intend-to-write lock should be shared, as we would like other concurrent transactions to be able to read the data item before it is actually written. The shared locks on the writeset can then be upgraded to exclusive locks at commit

time, and the values actually updated.

Thus our protocol is along the following lines:

- Get shared read locks on the read-only items, and shared intend-to-write locks on the write set.
- Read the readset into local storage and release the locks on the read-only items.
- Service the reads and writes issued by the transaction from and to local storage.
- Upgrade the shared locks on the writeset to exclusive locks and perform actual writes to the database during commit, and finally release all locks.

As mentioned earlier, the merits of this approach are the early release of shared read locks and short holding period of exclusive locks. The protocol is obviously non-two-phase, as some shared locks are released before some other shared locks are upgraded to exclusive locks. The problem with this initial scheme is that it can produce non-serializable schedules, and thus is unacceptable.

Now our aim is to use this basic idea, as described above, and add some checks to ensure serializability. We show that this is possible if we use some *marker locks* and a *validation phase*. The algorithms used to handle the marker locks are non-trivial and are described in detail in the following sections.

$\begin{matrix} \text{old} \rightarrow \\ \text{new} \\ \downarrow \end{matrix}$	WHITE	BLUE	GREEN	YELLOW	RED
WHITE	Y	Y	Y	Y	Y
BLUE	Y	Y	Y	Y	Y
GREEN	Y	Y	Y	Y	N
YELLOW	Y	Y	N	N	N
RED	Y	Y	N	N	N

Figure 2: Lock Compatibility Table (Note: The table is asymmetric).

2.2. Locking

The *Five Color* Protocol uses five types of locks, namely, *Green* (GL), *Yellow* (YL), *Red* (RL), *White* (WL) and *Blue* (BL). The compatibility matrix for these locks are shown in Figure 2.

The *Green* lock is a shared lock used for reading the read-only part of the readset. The *Yellow* lock is a shared intend-to-write lock and is stronger than the *Green* lock. It is used to lock the items in the writeset, as a preparatory measure, before they are actually updated. The *Yellow* lock is compatible with the *Green* locks, allowing a *Yellow* locked item to be read as a read-only data item by another transaction, but it is not compatible with itself, preventing simultaneous update attempts. (The *Five Color* protocol allows upgrading of *Yellow* to *Red*, but not *Green* to *Yellow* or *Red*.)

Note that a *Green* lock can be obtained on a *Yellow* locked item but a *Yellow* lock cannot be obtained on a *Green* locked item. This feature makes the compatibility matrix asymmetric. This asymmetry is not a major feature of the algorithm but has to be provided to prevent a particular race condition that can arise in the lock acquisition phase, described later.

The *Red* lock is the exclusive lock used for writing, and is compatible only with the *White* and *Blue* marker locks. Neither the *Green* locks nor the *Red* locks are held over extended lengths of time. Only *Yellow*, *Blue* and *White* locks exist nearly as long as the transaction does.

The *White* and *Blue* locks are the marker locks[†]. They are compatible with all other locks. These are used by transactions to keep track of data items read or written by other transactions and cause *triggering* as described later [Ca81]. Briefly, the semantics of the *White* and *Blue* locks are:

- A data item *x* is *White* locked by a transaction *T*, if *T* has read *x*, or if there is a transaction *T'*, such that *T'* must follow *T* in a serialization of the execution, and *T'* has read *x*.
- A data item *x* is *Blue* locked by a transaction *T*, if there is a transaction *T'*, such that *T'* must follow *T* in a serialization of the execution, and *T'* intends to write *x*.

2.3. Transaction Phases

A transaction *T* goes through several phases. When the transaction is initiated, (*arrival point*), the transaction manager obtains *Yellow* locks on the data items in $Wr(T)$ and *Green* locks on $Rd(T)-Wr(T)$ i.e. the *read-only* data items. This is the *lock acquisition phase*.

After all the locks are obtained, the transaction is considered for *validation*. If the transaction passes validation then it has to acquire some *Blue* and *White* locks. It gets *Blue* (and *White*) locks on data items written (and read) by some other concurrently running transactions. It also has to assign *White* and *Blue* locks on the items in its read and write sets to some other

[†]The reason for calling the *Blue* and *White* locks, "locks" and not just "markers" are as follows. Markers are used by agents to mark objects. Irrespective of how many times an object is marked, it becomes unmarked when the marker is removed. With a lock we can ask questions such as *Which agent holds locks on this object?* or *What are the objects locked by this agent?* Conceptually markers are too weak to answer both these questions.

transactions. This is called *lock inheritance phase*, and the exact details of which data items are locked are explained later. After completion of the lock acquisition phase, the transaction reaches its *locked point*.

Subsequently all the items in $Rd(T)$ are read into local storage, the *Green* locks are converted (downgraded) to *White* locks and the transaction enters the *processing phase*. Then the transaction commences execution (*start point*). When the transaction completes execution all *Yellow* locks are converted to *Red* locks. This is the *Final Locked Point*. The updated items are written to the database, all locks are released, and the transaction terminates. These phases and points are illustrated in Figure 3.

The following is an informal outline of how a transaction manager handles a transaction.

→ *Arrival Point* (Transaction T arrives)

- Get *Yellow* locks on $Wr(T)$ and compute Before/After sets (explained later)
- Get *Green* locks on $Rd(T) - Wr(T)$
- Do validation and lock inheritance processing (explained later)

→ *Initial Locked Point*

- Read values of $Rd(T)$ into local storage,
- Downgrade *Green* locks to *White* locks,
- Start transaction processing.

→ *Start Point*

- Let T commence processing,
 - if T issues $read(x)$, then return the value of x from local storage,
 - if T issues $write(x)$, then write x in local storage.
- Processing ends.
- Upgrade *Yellow* locks to *Red* locks,

→ *Final Locked Point*

- Write updated items to the database,
- Release all (*White*, *Blue*, and *Red*) locks held by T.

→ *Termination Point* (Transaction T terminates)

2.4. Transaction Manager Algorithms

A transaction is said to be *live* if it has arrived, but has not terminated. For each live transaction T , the Transaction Manager maintains two temporary sets, during lock acquisition phase, called $\text{Before}(T)$ and $\text{After}(T)$. These sets are accessed and updated only during the lock acquisition phase. The set $\text{Before}(T)$ is a set of transactions that are live, conflict with T , and must come before T in a serialization order. Similarly $\text{After}(T)$ contains those live and conflicting transactions that must come after T in the serialization order. The serialization order is determined by the actual order in which the locks are requested by the concurrent transactions. (Sometimes the *Before* and *After* sets contain a few recently terminated transactions, but that is of no consequence.)

As the transaction manager acquires locks on behalf of a transaction, it can determine which transactions must come before or after this transaction in the serialization order, by looking at the existing locks. These transactions are placed in the $\text{Before}(T)$ and $\text{After}(T)$ sets. The $\text{Before}(T)$ and $\text{After}(T)$ sets are constructed as follows.

Suppose T wants a *Green* lock on a data item x , and a set of transactions $\{T_i, T_j, \dots\}$ already possess *Blue* locks on x . The existence of *Blue* locks held by $\{T_i, T_j, \dots\}$ implies that some transaction(s) later than all of $\{T_i, T_j, \dots\}$ have written x . As T wants to read x , it must come after all of $\{T_i, T_j, \dots\}$. Thus $\{T_i, T_j, \dots\}$ must logically precede T , and they are added to $\text{Before}(T)$.

However, if some transaction T_i is holding a *Yellow* lock on x (when T is trying to *Green* lock it), this implies T_i will update x after T reads it. Hence T_i should come after T , and T_i is added to $\text{After}(T)$.

Similarly, during an attempt to *Yellow* lock an item x , all transactions holding *Blue* or *White* locks on x are added to $\text{Before}(T)$.

Validation is simply checking whether $\text{Before}(T) \cap \text{After}(T) = \emptyset$. If not, this implies that there are transactions that must come before as well as after T in the serialization order, and thus the resulting execution could be non-serializable. To prevent this from occurring, the transaction T is rescheduled. Avoiding livelocks (or starvation) due to rescheduling is dealt with in section 6.

If the transaction passes validation, then the *lock inheritance processing* has to be done. Some *Blue* and *White* locks are granted to various transactions as a result of lock inheritance. This is done as follows.

Suppose T has passed validation. For each transaction (τ) in $\text{After}(T)$, T is delayed until τ has reached locked point, and T is given *White* locks on all the data items *White* or *Green* locked by τ and *Blue* locks on all the data items *Blue* or *Yellow* locked by τ . Then all transactions in $\text{Before}(T)$ are given *White* locks on the readset of T , *Blue* locks on the writeset of T . Finally, all transactions in $\text{Before}(T)$ get *White* locks on all data items *White* locked by T , and

iii) Validation and lock inheritance:

```
if (After(T)  $\cap$  Before(T)  $\neq \emptyset$ )
    (* Validation *)
then RESCHEDULE T
else
    begin
        for all  $\tau \in$  After(T) do
            begin
                Wait for  $\tau$  to reach locked point,
                BLS(T)  $\leftarrow$  BLS(T)  $\cup$  BLS( $\tau$ )  $\cup$  Wr( $\tau$ );
                (* T gets Blue locks on the writeset of  $\tau$ 
                and all items Blue locked by  $\tau$  *)
                WLS(T)  $\leftarrow$  WLS(T)  $\cup$  WLS( $\tau$ )  $\cup$  Rd( $\tau$ )
                (* T gets White locks on the readset of  $\tau$ 
                and all items White locked by  $\tau$  *)
            end;
        end;

        for all  $\tau \in$  Before(T) do
            if  $\tau$  has not terminated then
                begin
                    BLS( $\tau$ )  $\leftarrow$  BLS( $\tau$ )  $\cup$  BLS(T)  $\cup$  Wr(T);
                    (*  $\tau$  gets Blue locks on all items
                    Blue locked by T and the writeset of T *)
                    WLS( $\tau$ )  $\leftarrow$  WLS( $\tau$ )  $\cup$  WLS(T)  $\cup$  Rd(T)
                    (*  $\tau$  gets White locks on all items
                    White locked by T and the readset of T *)
                end
            end;
        end;
    end;
```

We stress that there is no assumption of atomicity of any part of the above Transaction Manager algorithms, except in the test and set needed for the implementation of the LOCK function. The LOCK function is the standard locking primitive. If the lock requested cannot be granted due to a conflict, then the process is suspended (or queued) until the lock can be granted. These algorithms can be executed concurrently with all the activities of the other transaction on the database system, including lock acquisition phases of other transactions.

In our protocol, locks can be held only by uncommitted transactions. Hence if a transaction τ in Before(T) commits before T gets to do lock inheritance, then τ does not have to be

given any locks during the lock inheritance of T. Also, during lock inheritance, a transaction T gets *Blue/White* locks on items locked by transactions in $\text{After}(T)$. But all transactions in $\text{After}(T)$ are guaranteed to be live[†] when transaction T is in the lock inheritance phase (Lemma 3), and thus we do not have to keep records of the locks held by dead transactions.

2.5. Intuitive Discussion

The transaction manager of the *Five Color* protocol handles all the locking and concurrency control needed to run transactions. The transactions themselves do not have any knowledge of the protocols. A transaction has the following structure:

```
Begin-Transaction ( Readset, Writeset)
    ...
    { Statements }
    ...
End-Transaction.
```

The *Begin-Transaction* statement starts up the lock acquisition phase of the transaction. The transaction manager does the acquiring of locks (using the readset and writeset information provided as parameters) and then performs the validation and inheritance phases. After all the preprocessing is completed, the transaction starts executing the statements, needing no further assistance from the transaction manager. When the *End-Transaction* statement is reached, the transaction manager is called upon to do the *Yellow* to *Red* lock upgrades, actual writes and commit.

Though acquiring locks is done on behalf of the transaction, by the transaction manager, in the rest of our discussions we will refer to this event as “a transaction obtains a lock” because of simplicity and conceptual clarity.

As described in section 2.1, the basic algorithm that the *Five Color* Protocol uses is as follows. First, the writeset is locked using the *Yellow* lock. Then the readset is locked with *Green* locks. After the readset is read into local storage, the *Green* locks are downgraded to *White* locks.

After the locks are obtained, the validation and lock inheritance processing is done, the transaction commences execution. After the execution ends, (commit point) the *Yellow* locks are upgraded to *Red* locks, the data written out, and all locks released. The need for validation, and how it is done is discussed later in this section.

[†] In fact, Lemma 3 shows that the transactions in $\text{After}(T)$ will be “active” when T reaches locked point, which is a stronger condition than “live” during lock inheritance.

Suppose Transaction T_1 is running, and it has released all the *Green* locks on the readset. Now T_2 can update an item which was read by T_1 , making T_2 a logically *later* transaction in the serialization order. In two-phase locking T_2 has to wait until T_1 releases the read lock, and this may mean waiting until T_1 commits. If there are many reading transaction on database systems, transactions that write can be held up for long periods of time.

In addition, write locks held by transactions updating data items, cause delays for reading transactions. The *Yellow* locking scheme allows read-only transactions to read data items even in the presence of update transactions. Upgrading of locks are allowed in two-phase protocols but this often leads to trivial deadlocks [GrHoKoOb81, Ko83].

The early release of read locks may allow non-serializable executions, and this has to be avoided. Validation is used to avoid possible inconsistencies. The following example shows the need for validation.

Suppose T_1 is running and has read x and released the *Green* lock on x . T_1 has a *Yellow* lock on y which it intends to update later. Now a new transaction T_2 can choose to update x (get a *Yellow* on x), making T_2 a logically later transaction than T_1 . T_2 can also attempt to read (or get a *Green* lock) on y , an item that is *Yellow* locked by T_1 (making T_2 come logically before T_1). This would lead to a non-serializable condition

Early release of read locks and late acquisition of write locks as is done by the *Five Color* protocol leads to non-serializable executions, a simple case of which has been depicted above. The validation and lock inheritance algorithms have been designed to detect such situations and provide serializability. The algorithms use the following strategy.

A transaction T_1 holds *White* locks on all items it has read. T_1 also holds *White* locks on data items read by other concurrent transactions that should come after T_1 in the serialization order. Similarly, T_1 holds *Blue* locks on all items written (or to be written)[†] by concurrent transactions that should come after T_1 in the serialization order. (This property is proved later in Lemma 4). This implies T_1 “knows” about all the data items read or written by “later” transactions.

Suppose a new transaction T_2 arrives in this situation, and *Yellow* locks an item that is *Blue* or *White* locked by T_1 . This implies T_2 will update an item that has been read or written by some transaction that is after T_1 in the serialization order. Thus T_2 should be after T_1 in the serialization order. As expected, the *Yellow* locking of this item causes T_1 to be placed in $\text{Before}(T_2)$ (because T_1 is before T_2 in the serialization order). In fact all transactions such as T_1 which should come before T_2 get into $\text{Before}(T_2)$.

[†] The fact that T_1 holds *Blue* locks on data items that have not yet been updated can lead to an anomaly in certain situations. See section 4 for a modification that avoids this.

Now T_2 can cause a non-serializable condition by attempting to *Green* lock an item, which is already *Yellow* locked by a transaction such as T_1 . *Green* locking a *Yellow* locked item implies that T_2 is getting a view of the database as it was before T_1 updates the item it *Yellow* locked, thus T_2 should precede T_1 . But this action would cause T_1 to get into $\text{After}(T_2)$, and T_2 would fail validation.

The fact that each transaction T_1 has *White* and *Blue* locks on the read and write sets of all transactions that come after T_1 is ensured as follows: Whenever a new transaction, T_2 , acquires the *Green* and *Yellow* locks, its $\text{After}(T_2)$ set contains all concurrent transactions that should come after T_2 in the serialization order. T_2 then acquires *White* (and *Blue*) locks on the readset (and writeset) of all transactions in $\text{After}(T_2)$. Similarly, it gives all the transactions in $\text{Before}(T_2)$ *White* locks on all its *White* locked objects, and *Blue* locks on all its *Blue* locked objects and $\text{Wr}(T_2)$.

The maintenance of the *White* and *Blue* locks might seem contrived and unnecessary, but it has to be done to prevent conditions typified by the following example. Consider three transactions, T_1 , T_2 and T_3 having the following traces:

$$\begin{aligned} T_1 &: R_1(X) W_1(Y) \\ T_2 &: W_2(X) W_2(Z) \\ T_3 &: R_3(Y) W_3(Z) \end{aligned}$$

A particular sequence of execution under the *Five Color* protocol is depicted in Figure 4.

When T_3 arrives T_2 has terminated and T_1 is executing. Since T_2 has updated X , and T_1 has read an older value, T_2 must be after T_1 in the serialization order. The locking rules will allow T_3 to read Y as it existed before T_1 updates it, and update Z which has already been updated by T_2 . However, this is non-serializable. Since at this point T_2 is no longer alive, and T_1 does not touch Z , there seems to be no basis for disallowing T_3 from reading Y and writing Z , unless the *Blue* and *White* locks, and the *Before* and *After* sets are used.

When T_3 *Yellow* locked Z , Z was *Blue* locked by T_1 . This caused T_1 to become a member of $\text{Before}(T_3)$. When T_3 *Green* locked Y , since Y was *Yellow* locked by T_1 , T_1 became a member of $\text{After}(T_3)$. Now that the intersection of $\text{Before}(T_3)$ and $\text{After}(T_3)$ is not empty, T_3 fails validation, and the non-serializable execution is not allowed. This is an instance of *triggering* that has been mentioned in section 2.2 and discussed further in section 7.

3. Properties and Correctness

We now state the properties of the *Five Color* protocol and prove it achieves serializability. The following Lemmas are useful to understand the details of the algorithms, but the actual proofs may be skipped at the first reading. In order to show that the *Five Color* protocol assures serializability, we define a standard precedence relation \rightarrow based on the well known

T_1	T_2	T_3	Notes
Arrives, <i>Yellow</i> locks Y <i>Green</i> locks X reads X Converts <i>Green</i> on X to <i>White</i> .			
Gets <i>Blue</i> lock on X and Z	Arrives, <i>Yellow</i> locks X <i>Yellow</i> locks Z Updates X Updates Z Terminates		$T_1 \in \text{Before}(T_2)$
Executing		Arrives <i>Yellow</i> locks Z <i>Green</i> locks Y	$T_1 \in \text{Before}(T_3)$ $T_1 \in \text{After}(T_3)$ (fails validation, see discussion.)

Figure 4: Execution of the example history under the *Five Color* protocol.

notions of read-write and write-write conflicts [EsGrLoTr76, BeGo80]. The acyclicity of this relation implies serializability (though not vice versa [Pa79]). We will show that the *Five Color* protocol assures acyclicity of the \rightarrow relation.

All transactions that partake in the relations in the definitions are assumed to be transactions that have already passed the validation phase. Transactions that have not passed validation also generate relationships with other transactions, but since these transactions do not have any effect on the database, we choose to ignore them in the correctness proofs.

Definitions:

- A transaction is *active* if it has reached locked point, but has not completed acquiring *Red* locks.
- Define a binary relation (\rightarrow) over transactions. Two transactions T_i and T_j are related by the precedence relation ($T_i \rightarrow T_j$), if and only if:
 - i) T_i reads some data item x from the database, and at some later point T_j writes x into the database (r-w conflict), or
 - ii) T_i writes some data item x into the database, and at some later point T_j reads x from the database (w-r conflict), or
 - iii) T_i writes some data item x into the database, and at some later point T_j writes x into the database (w-w conflict).

The graph defined by the precedence relation is called the serializability graph [BeHaGo87].

Lemma 1

L.1: The relation $T_i \rightarrow T_j$ occurs if and only if one of the following cases take place:

- i) T_i gets a *Green* lock on x and then T_j gets a *Yellow* lock on x after T_i releases the *Green* lock. This arc is defined as of type $[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$.[†]
- ii) T_j gets a *Yellow* lock on x , then while T_j holds the *Yellow* lock, T_i gets a *Green* lock on x , and then T_j converts the *Yellow* lock into a *Red*. This arc is defined as of type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$.
- iii) T_i gets a *Yellow* lock on x , and later, after T_i unlocks x , T_j gets a *Green* lock on x . This arc is defined as of type $[\alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G]$.
- iv) T_i *Yellow* locks x , and later T_j *Yellow* locks x . This arc is defined as of type by $[\alpha \rightarrow \beta \mid \alpha Y, \beta Y]$.

Proof of L.1::

Simple, but lengthy.

■

The two cases ii) and iii) (above) look similar but cause very different results. In ii), T_j gets a *Yellow* lock on x and then, while the *Yellow* lock is held, T_i gets a *Green* lock on x . In

[†]The notations such as $[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$ denotes types of edges. If two transactions T_1 and T_2 are related as $T_1 \rightarrow T_2$, this relation can be caused in many ways. α stands for the transaction to the left of the \rightarrow symbol, and β is the transaction on its right. R, Y and G denote getting a *Red*, *Yellow* or *Green* lock. The sequence to the right of the '|' shows the sequence of lock acquisitions by α and β which led to the formation of the \rightarrow relation.

this case, T_j will update x , but T_i gets to see x as it existed before T_j updated it. Hence the serialization order of the transactions should be $T_i \rightarrow T_j$. However in case iii), under similar circumstances, if T_j gets a *Yellow* lock, and after this *Yellow* lock has been converted to *Red*, and released, T_i gets a *Green* lock on x , then the serialization order should obviously be $T_j \rightarrow T_i$. (The roles of T_i and T_j in the second example has been reversed, to be similar to the first example.)

The arcs of the serializability graph (\rightarrow) are caused by r-w, w-r and w-w conflicts. These conflicts happen when both the transactions have actually processed the conflicting reads and writes. However for ease of modeling we will assume that the arcs are created earlier. We define that an arc $T_i \rightarrow T_j$ is *created* when either T_i or T_j reaches locked point, whichever is later. (T_i and T_j must of course conflict as stated in Lemma 1.) This occurs after all the locks for T_i and T_j have been obtained, but before any actual conflict due to actual reading or writing has taken place. Also, if a transaction has not reached locked point, there is no arc either from or to it.

Note that the serializability graph thus formed is not identical to the graph formed by the r-w, w-r and w-w conflicts at some given point in time. This is because the arcs of the precedence graph are created prior to occurrence of the conflicts. Thus this serializability graph is in fact a superset of the conflict graph, which will become identical to the conflict graph when all activity on the database ceases. However our correctness criterion is the acyclicity of the conflict graph (see below) and acyclicity of the serializability graph implies acyclicity of the conflict graph.

Lemma 2

L.2: If $T_i \rightarrow T_j$ and T_j reached its *locked point* before T_i did, then the arc can only be of type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$.

Proof of L.2::

Since T_j has reached its *locked point*, it has obtained all its locks. The arc $T_i \rightarrow T_j$ could have been caused by four cases (Lemma 1). Consider each case separately:

$[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$

After T_i gets a *Green* lock on x , T_j cannot get a *Yellow* lock on x until T_i converts the *Green* lock to a *White* lock. Hence T_j cannot reach the *locked point* before T_i does and this case is impossible.

$[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$

This case is possible, as T_i may obtain a *Green* lock on x after T_j has placed a *Yellow*

lock on x .

$[\alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G]$

In this case T_i has to get a *Red* lock on x before T_j gets a *Green* lock on x . This makes it impossible for T_j to get to the *locked point* before T_i , thus this case is impossible.

$[\alpha \rightarrow \beta \mid \alpha Y, \beta Y]$

Again, T_i has to release the *Yellow* lock on x before T_j can place another *Yellow* lock on x . Hence T_j cannot reach its *locked point* before T_i , making this case impossible too.

Thus only the second case can take place, and hence the arc is of type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$. ■

Lemma 3

L.3: If $T_i \rightarrow T_j$ and T_j reached its *locked point* before T_i did, then T_j is still *active* when T_i reaches its *locked point*.

Proof of L.3

The arc $T_i \rightarrow T_j$ is of the type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$ (Lemma 2). Thus T_j gets a *Yellow* lock first, then T_i gets a *Green* lock and finally T_j upgrades its *Yellow* lock to *Red*. After T_i gets a *Green* lock on x , T_j cannot convert its *Yellow* lock to a *Red* lock until T_i converts its *Green* lock to a *White* lock. Thus when T_i reaches its *locked point*, T_j must be active. ■

Lemmas 2 and 3 show that unlike two-phase locking the serializability graph can grow “backwards” (see section 5). A transaction T_2 , which arrives later than transaction T_1 , may precede T_1 in the precedence order, if T_1 is active when T_2 arrives.

Lemma 4

L.4: If $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$ is a path in the serializability graph, and T_1 is active, then

L.4.1: T_1 possesses *White* locks on $Rd(T_n)$ (i.e. $Rd(T_n) \subset WLS(T_1)$), and

L.4.2: T_1 possesses *Blue* locks on $Wr(T_n)$ (i.e. $Wr(T_n) \subset BLS(T_1)$).

Proof of L.4.1:

Proof is by induction on N , the number of transactions in the path. We show it is true for $n = 2$, then assume it is true for $n = N$ and show it follows for $n = N + 1$.

$\boxed{n = 2}$

Let $T_1 \rightarrow T_2$, where T_1 is active. The arc in the path can be of four types. Consider each case separately:

$[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$

T_2 can obtain the *Yellow* lock on the data item (say x) after T_1 has converted the *Green* lock it was holding to a *White* lock. When T_2 gets a *Yellow* lock on x while T_1 holds a *White* lock on it, T_1 is added to $\text{Before}(T_2)$. Then T_1 inherits *White* locks on T_2 's readset. Hence $\text{Rd}(T_2) \subset \text{WLS}(T_1)$.

$[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$

When T_1 gets a *Green* lock on x while T_2 is holding a *Yellow* lock on x , T_2 gets into the set $\text{After}(T_1)$. Then T_1 inherits *White* locks on the readset of T_2 . Hence $\text{Rd}(T_2) \subset \text{WLS}(T_1)$.

$[\alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G]$

T_1 cannot be active in this case, as T_1 has released a *Red* lock before the edge could be created and hence has passed into inactive state.

$[\alpha \rightarrow \beta \mid \alpha Y, \beta Y]$

T_1 cannot be active in this case as T_1 has released the *Yellow* lock, which implies T_1 has upgraded the *Yellow* to *Red* and released the *Red* lock.

Thus T_1 has *White* locks on the readset of T_2 .

$\boxed{n = N}$

We assume Lemma 4 holds for some value N of n .

$\boxed{n = N+1}$

Consider a path consisting of $n (=N+1)$ transactions. By definition all the transactions in this path have reached their *locked points*. Let T_k , be the transaction that reached its locked point last, amongst all the transactions in the path:

$$T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_{k-1} \rightarrow T_k \rightarrow T_{k+1} \rightarrow T_{k+2} \rightarrow \dots \rightarrow T_n$$

Now there are three cases:

- 1) $2 < k < n-1$
- 2) $k=1$ or 2 or $n-1$ or n .

First consider case 1]:

Consider the instance when T_k just reached its *locked point*. At this point the arcs:

$$\begin{array}{l} T_{k-1} \rightarrow T_k \quad \text{and} \\ T_k \rightarrow T_{k+1} \end{array}$$

are created. Thus prior to this there were two shorter paths, each path having a length less than N , for which Lemma 4 holds.

$$\begin{array}{l} \text{a) } T_1 \rightarrow \dots \rightarrow T_{k-1} \quad \text{and} \\ \text{b) } T_{k+1} \rightarrow \dots \rightarrow T_n. \end{array}$$

For path a), T_1 is alive (by definition) and thus T_1 has *White* locks on $Rd(T_{k-1})$ and *Blue* locks on $Wr(T_{k-1})$, since $(k-1) < N$. As T_k reached *locked point* later than T_{k+1} , it follows that T_{k+1} was active when T_k reached *locked point*, and thus had *White* locks on $Rd(T_n)$, and *Blue* locks on $Wr(T_n)$ (because of path b).

By Lemma 2, the arc $T_k \rightarrow T_{k+1}$ must be of type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$. Hence T_k gets a *Green* lock on some x that is *Yellow* locked by T_{k+1} . Thus T_{k+1} is in $\text{After}(T_k)$, and since T_k gets *White* locks on all items *White* locked by T_{k+1} , T_k gets *White* locks on $Rd(T_n)$.

The $T_{k-1} \rightarrow T_k$ arc can be of four types. Treating them separately:

$[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$

In this case T_k sets a *Yellow* lock on a data item x , which is in $Rd(T_{k-1})$, and thus x is *White* locked by T_1 . This makes $T_1 \in \text{Before}(T_k)$, and thus T_1 inherits *White* locks on all $\text{WLS}(T_k)$, which contains $Rd(T_n)$. Thus T_1 gets *White* locks on $Rd(T_n)$.

$[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$

This cannot happen as T_{k-1} must reach *locked point* before T_k . For if T_k gets a *Yellow* lock, and then T_{k-1} gets a *Green* lock, T_k becomes a member of the set $\text{After}(T_{k-1})$. Now T_{k-1} has to wait for T_k to reach *locked point*, before it can reach *locked point*. (This is a condition during lock inheritance, please see the algorithm description at the end of section 2.3.)

$[\alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G]$

In this case T_k sets a *Green* lock on a data item x , which is in $Wr(T_{k-1})$, and thus x is *Blue* locked by T_1 . This causes $T_1 \in \text{Before}(T_k)$, and thus T_1 inherits *White* locks on all $WLS(T_k)$, which contains $Rd(T_n)$. Thus T_1 gets *White* locks on $Rd(T_n)$.

$[\alpha \rightarrow \beta \mid \alpha Y, \beta Y]$

In this case T_k sets a *Yellow* lock on a data item x , which is in $Wr(T_{k-1})$, and thus x is *Blue* locked by T_1 . This makes $T_1 \in \text{Before}(T_k)$, and thus T_1 inherits *White* locks on all $WLS(T_k)$ which contains $Rd(T_n)$. Thus T_1 gets *White* locks on $Rd(T_n)$.

Thus for all cases, T_1 has *White* locks on readset of T_n .

For the situations where $k = 1$, or 2 , or $n-1$, or n ; the above proof can be modified. We sketch the case for $k=1$, and leave the rest to the reader.

When $k=1$, we have one path before T_1 reaches locked point:

$$T_2 \rightarrow \dots \rightarrow T_n.$$

Since this path has less than N transactions, Lemma 4 holds; and T_2 has *White* locks on $Rd(T_n)$. It can then be shown that when T_1 reaches locked point (T_2 is active) T_1 inherits all the *White* locks from T_2 , making T_1 have *White* locks on $Rd(t_n)$.

■

Proof of L.4.2:

Similar to proof of L.4.1. Substitute *Blue* for *White* and writeset for readset in proof of L.4.1

■

above.

Lemma 4 shows the most important property of the *Five Color* protocol. This implies that if a transaction T_1 is active, it “knows” about the read and write set of all transactions that come after T_1 . This property is used to achieve serializability by causing a validation conflict when a cycle is created by some transaction.

Theorem 1:

The protocol ensures serializability.

Proof

We show that the serializability graph is acyclic. The proof is by contradiction. Assume there can be a cycle in the \rightarrow relation. Choose a minimal cycle:

$$T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_{k-1} \rightarrow T_k \rightarrow T_1$$

Assume that T_k is the transaction to reach its *locked point* last, compared to all the other transactions participating in the cycle. It will be shown that if T_k accessed data items in a manner that caused the cycle in the \rightarrow relation, then T_k would have been rescheduled at the validation phase.

T_k causes the creation of two arcs, which cause the cycle. They are the arcs:

A1: $T_{k-1} \rightarrow T_k$ and

A2: $T_k \rightarrow T_1$.

As T_k is the last transaction to reach its *locked point*, T_1 must have reached its *locked point* before T_k , hence (by Lemma 2) the arc $\langle T_k \rightarrow T_1 \rangle$ must be of type $[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$. Thus T_k gets a *Green* lock on a data item *Yellow* locked by T_1 . Hence $T_1 \in \text{After}(T_k)$

Also, by Lemma 3, T_1 must have been active when T_k reached its *locked point*. Thus by Lemma 4, T_1 has *White* locks on $\text{Rd}(T_{k-1})$ and *Blue* locks on $\text{Wr}(T_{k-1})$.

The arc $\langle T_{k-1} \rightarrow T_k \rangle$ could be of four types. Let us treat them separately:

$[\alpha \rightarrow \beta \mid \alpha G, \beta Y]$

In this case $T_1 \in \text{Before}(T_k)$ (see proof of Lemma 4). Thus $T_1 \in (\text{Before}(T_k) \cap \text{After}(T_k))$, and hence T_k should have been rescheduled at validation, and the cycle could not have resulted.

$[\alpha \rightarrow \beta \mid \beta Y, \alpha G, \beta R]$

This type of arc could have been caused only if T_k reached *locked point* before T_{k-1} , which is not the case.

$[\alpha \rightarrow \beta \mid \alpha Y, \alpha R, \beta G]$

In this case $T_1 \in \text{Before}(T_k)$ (see proof of Lemma 4). Thus again $T_1 \in (\text{After}(T_k) \cap \text{Before}(T_k))$ and T_k should have been rescheduled.

$[\alpha \rightarrow \beta \mid \alpha Y, \beta Y]$

In this case, again $T_1 \in \text{Before}(T_k)$. Rest as above.

Thus there can be no cycle in the \rightarrow relation under the protocol, and hence the protocol ensures consistency of updates. ■

4. A Modification

The algorithm as described has an undesirable feature. It does not hamper consistency but may cause more aborts than necessary. Consider the following situation:

- 1) T_1 gets *Green* lock on x
- 2) T_1 downgrades *Green* lock to *White* lock
- 3) T_2 gets *Yellow* lock on x

4) T_3 gets *Green* lock on x

At step 3, T_1 is in $\text{Before}(T_2)$ and thus gets a *Blue* lock on x . When T_3 gets a *Green* lock on x , $\text{Before}(T_3)$ contains T_1 . Actually T_1 and T_3 are unrelated. The anomaly exists as T_2 has *not yet written* x when T_3 reads x . There would have been no problem if T_2 had terminated before T_3 read x , and in this case T_3 would have to be after T_1 .

Thus in a path of transaction $T_1 \rightarrow T_2, \dots, \rightarrow T_k$, T_1 should have *Blue* locks only on those data items that have been updated by T_2, \dots, T_k , and not on all data items in the writeset of T_2, \dots, T_k .

The following is a very brief description of a method to prevent the above anomaly. Introduce another type of lock, called a *I-Blue* lock (or *Intent-Blue* lock). During lock inheritance, *Blue* locks are obtained on items already *Blue* locked by other transaction, while *I-Blue* locks are obtained on the writesets (uncommitted updates) of the transactions concerned. When a transaction commits, all the *I-Blue* locks held on its writeset by other transactions are changed to *Blue* locks. All other algorithms remain the same.

This modification is not incorporated in the algorithm as described in section 2. This is to avoid introduction of extra complexity, which has no bearing on the correctness of the protocol, and simplify understanding of the protocol. This is not a correctness issue, nor an important point in the concepts used in the *Five Color* protocol.

5. Deadlocks

In the *Five Color* protocol there is potential for two types of deadlocks, one due to locking and the other due to waiting for other processes to reach their locked points. Let us address them separately.

The first form of deadlock is the one encountered in traditional locking protocols, and is caused by transactions in a circular wait, trying to obtain locks. This form of deadlock can be prevented in this protocol, using the predeclared read and write sets. We define an ordering of resources, and acquire all locks in an increasing order of resources. First all the *Yellow* locks are obtained, in increasing order, and then all the *Green* locks are obtained in that order. The *Yellow to Red* conversion is also done in the increasing order of resources (data items).[†]

Theorem 2

The pre-ordered locking strategy used in the *Five Color* protocol is deadlock free.

[†] If we use only one type of (exclusive) lock, this strategy of obtaining locks in a predefined fashion is known to be deadlock free. However it is not true in general, where several types of locks with various compatibilities are used. In our case, however, it is deadlock free, as shown in theorem 2.

Definition

Suppose transaction T_i requests a lock on item x in mode m_1 , and item x is already locked in mode m_2 by transaction T_j . If mode m_1 is incompatible with mode m_2 we say T_i *waits-for* T_j . The waits-for relation is denoted as $T_i \xrightarrow{w} T_j$. The directed graph defined by the waits-for relation over all transactions is called the waits-for graph.

Proof

It can be shown that a cycle in the waits-for graph is an instance of a deadlock condition. Suppose deadlocks can take place in our system. Consider an instance of a deadlock involving k transactions, with the cycle in the waits-for graph being:

$$T_1 \xrightarrow{w} T_2 \xrightarrow{w} \dots \xrightarrow{w} T_k \xrightarrow{w} T_1$$

Step i)

Suppose none of the transactions T_1 to T_k are in the process of acquiring *Red* locks, that is they are in the *Green* or *Yellow* locking phase. Now T_1 to T_k cannot be all in the *Green* lock or *Yellow* lock acquiring phase. (If there is one type of locking, this strategy is known to be deadlock free). Thus some transactions are waiting for a *Green* lock, and others are waiting for a *Yellow* lock. Hence at some point, a transaction T_i is waiting for a *Green* lock on a data item x , for which T_j is holding a *Yellow* lock. This is a contradiction as the *Green* lock is compatible with an existing *Yellow* lock. Thus at least one transaction must be holding a *Red* lock.

Step ii)

Suppose one of the transactions, say T_1 , is in the process of upgrading its *Yellow* locks to *Red* locks, and holds at least one *Red* lock. A transaction which is trying to upgrade a *Yellow* lock on x , to a *Red* lock will have to wait only if some other transaction holds a *Green* lock on x (as no other transaction can hold a *Yellow* lock on x). Thus if T_1 is waiting for T_2 , then T_2 must be holding a *Green* lock. Since T_2 is also waiting, it must be waiting to get another *Green* lock, since acquiring of *Green* locks are done after acquiring of *Yellow* locks, and all *Green* locks are released before any *Red* locks are acquired. Thus T_2 is in the phase of acquiring *Green* locks.

Similarly, a transaction trying to acquire a *Green* lock on x will wait for another transaction only if that transaction holds a *Red* lock on x , as *Red* is the only lock incompatible with the *Green* lock.

Thus if $T_1 \xrightarrow{w} T_2 \xrightarrow{w} \dots \xrightarrow{w} T_k \xrightarrow{w} T_1$ is a cycle of waiting transactions and T_1 is in the *Red* lock acquiring phase, then so is $T_3, T_5 \dots$, and $T_2, T_4 \dots$, are in the *Green* lock acquiring phase (that is, the cycle comprises only of transactions in the *Green* or *Red* lock acquiring

phases).

The proof that there cannot be a cycle in a set of transactions having the above properties, is very similar to the proof that there cannot be a cycle in a set of transactions acquiring exclusive locks in a predefined order, and is not included here for brevity [Ha68].

■

The second form of deadlock is peculiar to this protocol. This form of deadlock involves one or more transactions in the lock inheritance phase. Note that in the lock inheritance phase, a transaction T has to wait for all transactions in $\text{After}(T)$ to reach locked point. This can cause deadlocks. The following is an example of a deadlock caused by two transactions in the lock inheritance phase.

- i) T_1 *Yellow* locks x
- ii) T_2 *Yellow* locks y
- iii) T_1 *Green* locks y
- iv) T_2 *Green* locks x

In this sequence of events the following problem takes place. Steps i) and ii) cause no surprises, but in Step iii) as T_1 tries to *Green* lock y , which is *Yellow* locked by T_2 , T_2 becomes a member of $\text{After}(T_1)$. Similarly, when T_2 *Green* locks x , T_1 becomes a member of $\text{After}(T_2)$.

Now both transactions will pass the validation, and wait for all transactions in their After sets to reach locked point. T_1 will thus wait for T_2 to reach locked point before it can reach locked point, and vice versa. Thus we have a deadlock. This deadlock effectively prevents the non-serializable log that could result if the transactions were allowed to continue.

The deadlock can involve transactions waiting for locks as well as transactions waiting for other transactions to reach locked point. However as proved above there are no deadlocks involving only transactions waiting for locks.

The deadlocks can be detected by standard deadlock detection algorithms, or by timeouts. As the deadlock occurs before the transactions start execution, there is no rollback involved and the overhead suffered is small.

Since the transactions which participate in deadlocks do not do any processing, we believe timeouts may be an easier and more efficient method to deal with deadlocks. Lack of processing means there are no processing delays and the timeouts can be fine tuned better to take into account the locking delays and cause deadlock warnings if something takes too long to happen.

Every deadlock cycle has at least one transaction waiting for the completion of the lock acquisition phase of another transaction. We propose that this is the point where a timeout should be introduced. The delay for the timeout can be a function of the number of locks the

other transaction has to acquire. This will keep the probability of detection of false deadlocks to be low.

6. Livelocks

There is a small chance of livelocks or starvation in this protocol. This is due to the rescheduling of transactions because of validation failure. There is no guarantee that an aborted transaction will finally be able to run. However we feel that the chances of starvation is extremely small. But low probability of starvation is still not a guarantee against starvation, so we propose the following method of avoiding livelocks.

If a transaction gets aborted due to validation failure a large number of times, we label the transaction as *starvation prone*. The writeset of a starvation prone transaction is then expanded to contain its readset. The transaction thus acquires only *Yellow* locks during lock acquisition, and as a result has an empty After set. This transaction is guaranteed to pass validation, avoiding the livelock problem. Also it never waits for any other transaction to complete lock inheritance. It may still, however participate in deadlocks. But note that to detect the deadlocks we are timing out and aborting a transaction that waits for transactions in its After set. Since the starvation prone transaction has an empty After set it will not get aborted even if it participates in a deadlock. Thus it is guaranteed to run to completion.

7. Discussion

As stated earlier, The *Five Color* protocol differs significantly from the two-phase locking protocol in the way the serializability graph may grow. In the two-phase locking protocol the precedence arcs are created when a transaction locks a data item. When a transaction locks an item (or upgrades a lock) it places itself after some other transaction, never before. Thus we say the serializability graph grows only in the *forward* direction. In fact this is the property of the two-phase locking protocol that ensures serializability.

Thus, in our protocol, a path under the precedence order can grow in *both* directions. Suppose transaction T has reached its locked point and possesses a *Yellow* lock on x. A new transaction τ arrives and gets a *Green* lock on x. When τ reaches locked point, the arc $\tau \rightarrow T$ is created. Now, even after T terminates, as long as τ is active, τ' may come and place itself before τ , and hence before T. In this way, it is possible for future transactions to be logically placed in the past.

Some locking protocols for example the Bayer et. al. protocol and the 2V2PL protocol discussed in section 1.4 allow one transaction to read an older value (before value) of a data item that is being updated by another transaction. This situation is the same as setting a *Green* lock on a *Yellow* locked item. However there are some important differences.

Since the Bayer and 2V2PL protocols are two-phase, they do not allow the serializability graph to grow in both directions. A transaction T_1 may read the before value of X while T_2 is updating X , but in this case T_2 must wait for T_1 to commit before it can commit. The Five Color protocol does not place this restriction.

Multiversion timestamp protocols and the Sterns et. al. protocols also allow transactions to read before values. It is difficult to compare these protocols with the Five Color protocol as the mechanisms used by them are quite different.

Thus the basic mechanism by which two-phase locking prevents serializability is not present in our protocol. Serializability is ensured, in this case by the *Blue* and *White* locks, and the validation procedure. Intuitively, if $T_1 \rightarrow \dots \rightarrow T_2$ is a chain of transactions, then T_1 "knows" about $Rd(T_2)$ and $Wr(T_2)$, because it has *White* and *Blue* locks, respectively, on these data items. If any transaction τ attempts to read any data item in $Wr(T_2)$ (or write any item in $Rd(T_1)$) then due to the "triggering" caused by *Green* (*Yellow*) locking of a *Blue* (*White*) locked item, T_1 becomes a member of $Before(\tau)$ and T_1 inherits *White* (*Blue*) locks on $Rd(\tau)$ ($Wr(\tau)$). Thus information about the read and write sets flow up a chain in the form of "inherited" *White* and *Blue* locks. Now if τ may cause a cycle in the \rightarrow relation by attempting to read an item *Yellow* locked by T , then T would become a member of $After(\tau)$ and violate the validation constraint. The other cases of information flow when the chain grows in the reverse direction, or when two chain as concatenated by a transaction is similar and is covered in the proofs (Lemma 4).

This property of the *Five Color* protocol allows the protocol to produce non-two-phase histories. For example, the *Five Color* protocol allows non-strictly-serializable[†] histories [BeGo80]. The following is an example of a non-strictly-serializable history that is allowed by the *Five Color* protocol. Some timestamp based protocols allow the following log, if the timestamp ordering is $T_3 < T_2 < T_1$.

Log	$R_1(x)$	$R_2(y)$	$W_1(y)$	$R_3(z)$	$W_2(z)$
Serial order:	T_3	\rightarrow	T_2	\rightarrow	T_1

Note: Though T_1 completes execution before T_3 commences, T_3 precedes T_1 in the serialization order.

The response of the protocol to this log is left to the reader. The following are the key

[†] A log L is strictly-serializable if there exists a serial order L_s of L such that if T_1 and T_2 are in L and their action do not interleave, then T_1 and T_2 appear in the same order in L as in L_s . A log that is serializable, but does not have the above property is non-strictly-serializable.

properties of the *Five Color* protocol. Some of these properties may lead to a higher concurrency level for the *Five Color* protocol, than what is possible for the family of two-phase protocols. Since the sets of histories produced by the *Five Color* protocol and the two-phase locking protocols are incomparable (that is, there exist histories produced by two-phase locking that are not produced by *Five Color* and vice versa), we cannot conclusively argue superiority of one over the other.

- Early release of read locks.

The *Five Color* protocol releases read locks as soon as the data is read. This allows other transactions to update these data items without having to wait for the reading transaction to commit. [In practical two-phase protocols, such as 2V2PL, all locks are held till commit point].

- Allowing reading of data items to be written later.

This allows reading transactions faster access to data items that would have been exclusively locked for quite some time otherwise. The Bayer protocol, Sterns protocol, 2V2PL and some other protocols have this feature. However, in all locking protocols, the updating transaction, then has to wait for the reading transaction to commit before it can commit.

- Non-compatibility of *Yellow* locks.

This property prevents some deadlocks. Consider the 2V2PL protocol. If two protocols issue read requests on X and then issues write requests on X, they will both try to upgrade the read locks they possess into write locks, causing a deadlock. This is termed a *trivial deadlock* [Ko83]. The deadlock is trivial to detect but just as serious as any deadlock, as one of the transactions have to be aborted. The incompatibility of the *Yellow* lock, avoids chances of trivial deadlocks and causes delay of an update transaction while another is updating the data item (delay is better than deadlock). A simulation study we conducted showed that trivial deadlocks are the most common form of deadlock in 2V2PL type protocols. However, it must be noted that the *Five Color* protocol avoids this problem since it has the knowledge of read/write sets, which 2V2PL does not.

- Preordering of locking requests.

This avoids deadlocks due to locking. The avoidance of such deadlocks as well as trivial deadlocks may cause the *Five Color* protocol to have a lower chance of deadlocks than protocols using un-ordered locking. However, any protocol that has knowledge of the read and write sets is able to do ordered locking. When attempting to do ordered locking, care has to be taken to see that locks are not held longer than necessary. The *Five Color* protocol has been designed with that in mind.

7.1. Conclusions

We have presented a locking protocol that uses an unconventional locking strategy, and knowledge about the read and write sets of the transactions to allow non-two-phase locking on a general database. We show that this protocol ensures serializability and has properties that may allow it to perform better than the two-phase locked protocols.

The *Five Color* protocol is substantially more complicated than the two-phase locking protocols. In fact the simplicity and elegance of the two-phase locking protocols are their major attractions. Nevertheless, we believe that the *Five Color* protocol is worth serious consideration.

Thus we conclude that it is possible for the *Five Color* Protocol to achieve more concurrency by anticipating the *absence* of conflicts in a large number of cases, due to the information available to the transaction manager about the readsets and writesets of the transaction.

8. Acknowledgements

We would like to acknowledge the anonymous referees for their in depth comments and for pointing out several shortcomings that escaped our notice. We would also like to thank Phil Bernstein and Magdi Morsi for their help and comments.

9. Bibliography

- [BaheRe80] Bayer, R. Heller, H. and Reiser, A. *Parallelism nad Recovery in Database Systems*. ACM Transactions on Database Systems, June 1980, pp. 139-156.
- [BeGo81] Bernstein, P.A. and Goodman, N. *Concurrency Control in Distributed Database Systems*. ACM Computing Surveys, June 1981, 13:2, pp. 185-222.
- [BeGo82] Bernstein, P.A. and Goodman, N. *Concurrency Control Algorithms for Multiversion Database Systems*. Proc. ACM SIGACT/SIGOPS Symp. on Principles of Distributed Computing, (1982)
- [BeGo83] Bernstein, P.A. and Goodman, N. *Multiversion Concurrency Control - Theory and Algorithms*. ACM Transactions on Database Systems, Dec. 1983.
- [BeHaGo87] Bernstein, P. A., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.
- [BeShRo80] Bernstein, P.A., Shipman, D.W., and Rothnie Jr., J.B. *Concurrency Control in a System for Distributed Databases (SDD-1)*. ACM Trans. on Database Systems 5:1, pp. 18-51 (1980).
- [BeShWo79] Bernstein, P.A., Shipman, D.W., and Wong, W.S. *Formal Aspects of Serializability in Database Concurrency Control*. IEEE Trans. on Software Eng., BSE-5:3, pp. 203-216 (1979).

- [BuSi83] Buckley G. and Silberschatz A., *Obtaining Progressive Protocols for a Simple Multiversion Database Model*, 9th International Conference on Very Large Databases, October 1983.
- [Ca81] Casanova, M.A. *The Concurrency Control Problem of Database Systems*. Lecture Notes in Computer Science, Vol. 116, Springer-Verlag, 1981.
- [Da83] Date, C.J. *An Introduction to Database Systems*, Vol 2., Addison-Wesley (1983).
- [DaKe83] Dasgupta, P. and Kedem, Z. M. *A Non-two-Phase Locking Protocol for General Databases*. (extended abstract), 9th international Conf. on Very Large Data Bases, Oct 1983.
- [Da84] Dasgupta, P. *Database Concurrency Control: Versatile Approaches to Improve Performance*. Ph.D. dissertation, State Univ. of New York, Stony Brook, Sept. 84.
- [EsGrLoTr76]
Eswaran, K.E., Gray, J.N., Lorie R.A., and Traiger, I.L. *On notions of Consistency and Predicate Locks in a Database System*, Comm. ACM 14:11, pp. 624-634 (1976).
- [FiMi82] Fischer, M.J. and Michael, A. *Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network*. Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, (1982).
- [Ga83] Garcia-Molina, H. *Using Semantic Knowledge for Transaction Processing in a Distributed Database*. ACM Trans. on Database Systems, pp. 186-213, (1983).
- [GaWi82] Garcia-Molina H. and Wiederhold G. *Read-Only Transactions in a Distributed Database*. ACM Transactions on Database Systems, 7:2, June 1982, pp 209-234.
- [Gr78] Gray, J.N. *Notes on Database Operating Systems*. IBM Research Report RJ2188 (1978).
- [GrHoKoOb81]
Gray J. N., Homan P., Korth H. and Obermark R. L., *A Straw Man analysis of the Probability of Waiting and Deadlock in a Database System*. Tech Report RJ 3066, IBM San Jose, CA 1981.
- [GrLoPuTr76]
Gray J., Lorie R.A., Putzolu F., Traiger I. *Granularities of Locks and Degrees of Consistency in a Shared Database*. in *Modeling in Data Base Management Systems*, Nijssen, G.M. (ed.), North Holland Publishing (1976).
- [Ha68] Havender J. W., *Avoiding Deadlocks in Multiuser Systems*, IBM Systems Journal, Vol. 7., No. 2, pp 74-84, 1968.
- [KaPa81] Kanellakis, P.C. and Papadimitriou, C.H. *The Complexity of Distributed Concurrency Control*. Proc. 22nd FOCS, pp. 185-197 (1981).

- [KeSi79] Kedem Z., Silberschatz A. *Controlling Concurrency Using Locking Protocols*. Proc. 20th IEEE FOCS, pp. 274-285 (1979).
- [KeSi81] Kedem Z., Silberschatz A. *A Non-two-Phase locking Protocol with Shared and Exclusive Locks*. Proc. Conference on Very Large DataBases, Oct'81
- [KeSi82] Kedem Z., Silberschatz A. *A Family of Locking Protocols Modeled by Directed Acyclic Graphs*. IEEE Trans. Software Engineering, (1982) pp. 558-562.
- [Ko83] Korth H., *Locking Primitives in a Database System*. Journal of the ACM, 30:1, January 1983, pp. 55-79.
- [KuRo81] Kung, H.T. and Robinson, J.T. *On Optimistic Methods for Concurrency Control*. ACM Trans. on Database Systems 6:2, pp. 213-226 (1981).
- [Li79] Lindsay B.G. et al., *Notes on Distributed Database Systems*. IBM Research Tech. Report RJ2571(33471) August 14, 1979.
- [Pa79] Papadimitriou, C.H. *The Serializability of Concurrent Database Updates*, J. ACM 26:4, pp. 631-653 (1979).
- [PaBeRo77] Papadimitriou, C.H., Bernstein P. and Rothnie, *Computational Problems Related to Database Concurrency Control*, Proc. of the Conference on Theoretical Computer Science, Waterloo, Canada, 1977, pp 275-282.
- [PaKa82] Papadimitriou, C.H. and Kanellakis, P.C. *On Concurrency Control by Multiple Versions*. Proc. ACM Symp. Principles of Database Systems, March 1982.
- [RoStLe78] Rosenkrantz, D.J., Stearns, R.I, and Lewis II, P.M. *System Level Concurrency Control for Distributed Database Systems*. ACM Transactions on Database Systems, 3:2 pp. 178-198 (1978).
- [SiKe80] Silberschatz, A. and Kedem Z.M. *Consistency in Hierarchical Database Systems*. J. ACM, 27:1, pp. 72-80 (1980).
- [StRo81] Stearns, R. E. and Rosenkrantz, D. J. *Distributed Database Concurrency Controls using Before Values*. Proc. Conference on Very Large DataBases, Oct'81.
- [Ul82] Ullman J.D. *Principles of Database Systems*., 2nd ed., Computer Science Press (1982), Potomac MD.
- [Ya82] Yannakakis, M. *A Theory of Safe Locking Policies in Database Systems*, J. ACM 29:3, pp. 718-740 (1982).

OPSLA '86

OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS

C O N F E R E N C E P R O C E E D I N G S

S E P T E M B E R 2 9 -
O C T O B E R 2 , 1 9 8 6

P O R T L A N D , O R E G O N

EDITED BY NORMAN MEYROWITZ
SPECIAL ISSUE OF SIGPLAN NOTICES
VOLUME 21, NUMBER 11, NOVEMBER, 1986



Sponsored by the Association for Computing Machinery.

A Probe-Based Monitoring Scheme for an Object-Oriented, Distributed Operating System

Partha Dasgupta[†]

Georgia Tech

Introduction

This paper presents an integration of three topics, namely distributed systems, object-oriented design and system monitoring. System monitoring is a functionality that enables a large system to keep track of the health of its individual components, especially if it has multiple instances of each type. This is particularly interesting in distributed systems that implement fault tolerance. Given the ability to detect failing, flaky or failed components (software modules and hardware units) the system has the ability to reconfigure the healthy units, on the fly, to work around the faulty ones.

The basic mechanism we propose, for monitoring, is the usage of *probes*. Probes are a powerful tool in many environments and has been proposed for deadlock detection, debugging, backup processing and so on. [ChMi82] Although the usage of probe-based algorithms can be generalized for many applications, the usage is very dependent on the exact semantics of the probe implementation and the environment in which they function.

In the following pages we explain the use of probes in a particular distributed system framework. The system

under consideration is a prototype of a distributed operating system and programming environment we call *Clouds*. *Clouds* is a object-oriented system building concept, that is wholly structured on the object concept. This forms a solid base layer for further research into the programming and environment aspects of object-oriented design.

We first present a short summary of the design criteria, goals and architecture of the *Clouds* system. This will illustrate the framework of fault tolerant distributed system that we have currently. We then present a design for the probe system that will fit elegantly into the existing design. We show how the probes (in this context) can be effectively used for system monitoring, reconfiguration and fault tolerance. The probe system also has some other payoffs like the easy implementations of interactive debugging support.

2. The *Clouds* Operating System

The *Clouds* operating system is a distributed, object-oriented operating system that supports objects, nested actions, reconfiguration, fault tolerance, orphan and deadlock detection and integration (location transparency). [Mo81, Al83, Mc84a, DaLeSp85]

The basic building blocks in *Clouds* are objects, actions and processes. Processes are carriers of the thread of control, on behalf of actions. The actions are atomic units of activity, consisting of a partial order of invocations of operations defined in objects (see sections 2.2 and 2.3).

2.1. The *Clouds* Architecture

The architecture of a distributed system can be partitioned into two main areas: the hardware configuration and the operating system structure. The operating system structure is loosely described above, as the set of objects, actions and processes. The hardware organization of *Clouds* is a set of processors, loosely coupled over a medium-to-high-speed network. The prototype configuration is shown in Fig. 1.

Author's Address:

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
ph: +1 (404) 894 2572

Electronic Address:

CSNet: partha@GaTech.CSNet
Arpa: partha%GaTech.CSNet @ CSNet-Relay.ARPA
UUCP: ...!akgua,allegra,amd,hplabs,lhnp4!lgatech!partha

Research has been supported in part by NASA under grant NAG-1-430.

Permission to copy without fee all or part of this material is granted provided that the copyright notice and the title of the publication and its date appear, and that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and permission.

CM 0-89791-204-7/86/0900-0057 75c

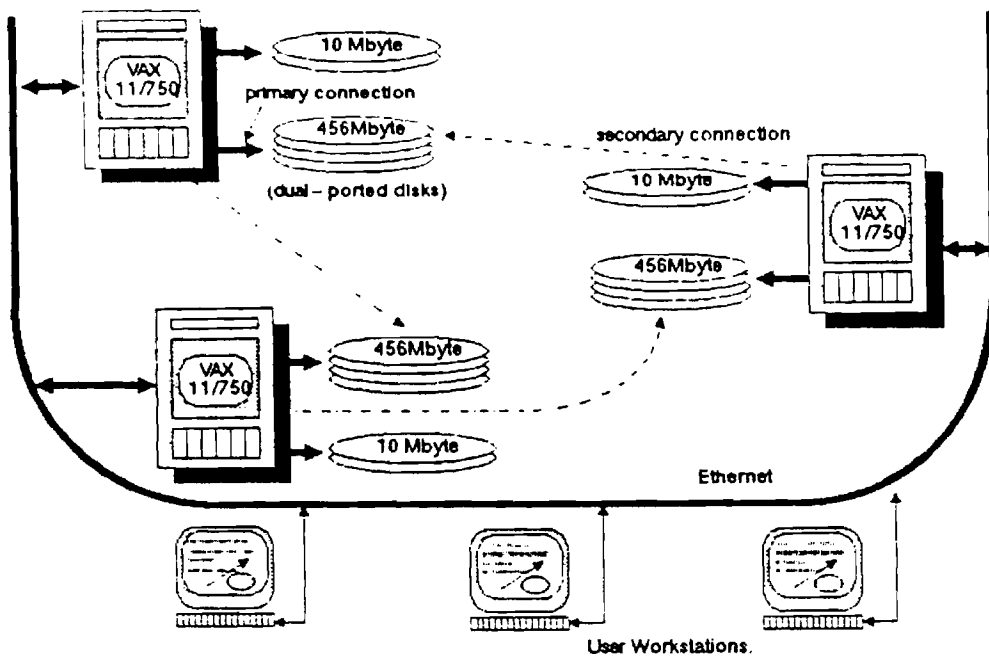


Fig. 1: The *Clouds* Prototype Configuration.

The prototype consists of three VAX/750 computers with 3 Meg memory each, connected by an Ethernet. The Ethernet serves both as a back-end network that links the machines of the distributed system, as well as a front end network that the users use to gain access to the system. The front end network is the gateway to the *Clouds* system, and the user access is through workstations (diskless SUN's or IBM-PC/AT's). Since the users are not hard wired to any site, in case of failures, users can be virtually transparently *floated* to another serviceable processor. The desktop computers used by the user run a special interfacing software system which can communicate to *Clouds* and can intelligently take part in system monitoring and can decide to change hosts if necessary, in cooperation with the surviving part of *Clouds*. [Mc84a]

The disk drives used for secondary permanent storage are dual ported. This allows reassignment of disk drives in case of processor failure and helps in reconfiguration, and leads to high availability of data.

2.2. Objects

The operating system structure of *Clouds* is based on the action/object paradigm. All permanent system components in *Clouds* are *objects*. Objects form a clean conceptual encapsulation of data and programs. They are useful in providing synchronization and recovery as will be described later. The objects are accessed by *processes* on behalf of actions.

In a simplistic view, an object is an instance of an abstract data type (cosmetically similar to *modules* in Modula-2 or *classes* in Simula). The object encapsulates permanent data and a set of routines that can access (read or update) the data. The only access path to the data contained in the object is through the routines (or operations) defined in the object. To the external world, the object is thus an entity providing a set of entry points. [Jo79]

Object instances in *Clouds* are permanent, that is they exist after they are created and are visible until they are explicitly deleted (like files). Any update caused on the data in an object by an operation invocation is also permanent. Thus all programs, data and files in the *Clouds* system are objects. For example, a file object is an object containing the file data, along with the read and write operation defined on the data. Thus each object can be tailor made to the application that requires the use of the object, and also tailored to suit the access methods and other update criteria applicable to the data contained in the object.

Clouds object are more powerful than just a black box containing some procedures and static data. It also contains a stack (temporary data), heap (permanent dynamically allocated data), and powerful support for concurrency control and recovery.

Objects in *Clouds* are passive, that is they consist of just a memory image, without any processes/servers associated with them. This allows virtually unlimited object instances with little overhead. Also invoking an object does not involve scheduling server processes. Neither is the concurrency limited by the degree of multi-threading feasible in the server.

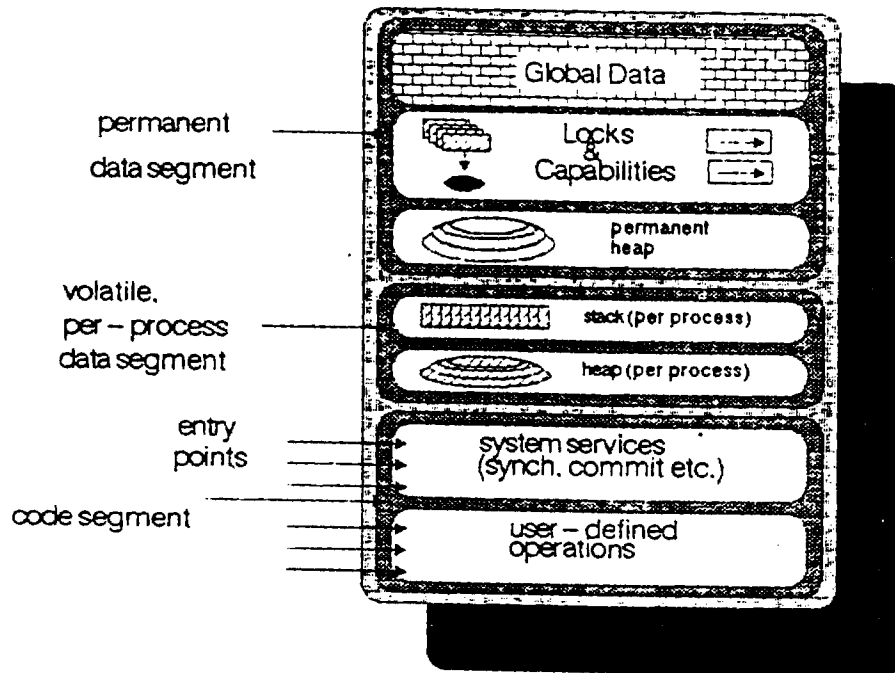


Fig. 2: Clouds Object Structure.

When a process invokes an operation on an object, it carries its thread of execution into the procedure defined in the object. Concurrently running processes thus are able to invoke concurrent operations in an object. This may or may not be allowable, depending upon the nature of the data in an object. The synchronization rules can be effectively built into the object (see section 2.4).

After a process updates the object, the updates may need to be rolled back if the transaction fails, or rather, the updates should be made after the transaction commits. The recovery features that ensure proper atomic updates can also be built into the objects.

The implementation techniques for synchronization and recovery are briefly explained in section 2.4. An outline of a Clouds object is shown in Fig. 2.

2.3. Actions

A user on the Clouds system can start up a transaction. A user transaction is a *top-level action*. A top level action is an atomic unit of work, that either effectively terminates and causes permanent updates to recoverable objects, or does not leave a trace.

A top level action can spawn more actions or subactions. These are nested actions. They can run concurrently with other subactions and the top level action, or they can be spawned sequentially. The subactions inherit the locks, and the views of the parent action, but execute independently. The details of the nested action semantics are omitted here.

The action management system is a part of the Clouds kernel that ensures the atomicity of the nested action scheme. The implementation of the action management system is an integral part of the kernel and is fast, efficient, and runs with very low overhead [Kø85]. Details about the management scheme and action semantics are omitted here.

2.4. Synchronization and Recovery in Clouds

Concurrency control (synchronization) and action atomicity are the responsibility of the objects touched by an action. In Clouds the concurrency control and recovery can be provided by default, by the Clouds system (auto-sync and auto-recovery) or can be tailored for specific objects by the application programmer, or can be omitted altogether for certain objects.

When an object is defined as *auto-sync*, the object compiler includes default code in the object entry and exit points to adhere to the 2-phase locking protocol. Each operation is classified as read or update operation depending upon the semantics of the operation. Invoking a read operation causes the invoking action to acquire a read lock on the object (if possible, or the process waits until the lock is obtained.) Similarly invoking a write operation causes the acquisition of a write lock, or the upgrade of a pre-existing read lock to a write lock.

Locks are not released when the operation returns or terminates, but are released by the commit phase, which is also handled by the object as described below.

When an action invokes an operation on a recoverable object, the action management system makes a bookkeeping entry to that effect. Each object, by default has three well known entry points, labeled *pre-commit*, *commit* and *abort*. When an action terminates, the action management system invokes these entry points to inform each object that participated in the computation, the result of the action. (The two commit entry points are needed by the 2-phase commit protocol, that is used to terminate successful actions). Activation of the commit or abort entry points in an object releases all the locks.

When a process invokes an object, the object is mapped to virtual memory, and all updates occur in memory. The abort operation simply frees the object, the pre-commit and commit pair flushes all the updates to the permanent storage. [McA182, PiSp85, Sp84]

2.5. Failure Resilience

Clouds also provides support for single faults, and crashes. A fault generally causes all affected action to abort, providing a guard against inconsistent executions caused by faults. Failure or network partitions can give rise to orphans that hold locks and cause loss of throughput. *Clouds* supports a time driven orphan elimination scheme that effectively hunts down and aborts all orphaned processes within a short interval after the orphanation [McH86]. The orphan detection routine makes use of a clever timeout scheme that is rarely affected by system sluggishness. The same scheme with minor modification doubles as a deadlock detector that breaks deadlocks.

The reconfiguration and monitoring system achieves high system resource availability. Since all the users interact with *Clouds* through an Ethernet, no user is hard wired to a site. A user has a primary site and a backup site associated with the user session. In case of a failure of the primary site, the backup site provides the user with system response. Thus site failures do not cause users to be left without access to the system. Since disks in the system are dual ported, they are accessible from two sites. Only one of the ports is active at any time. If the primary site controlling the disk fails, the other site activates the second port, and effectively all objects located at the faulty site get transferred to the working site. This also enhances the availability of the permanent data in the face of site failures.

The details of the algorithms that control the reconfiguration system are omitted for the sake of brevity.

3. Enhancing Fault Tolerance

The basic fault tolerance mechanism supported by *Clouds* is the action paradigm implemented by the action management system. The action paradigm ensures consistency of the computing environment in the face of failures. It is a backward recovery scheme. A failed action

causes an implicit rollback, and the action may not be able to execute until the fault has been rectified. This degree of fault tolerance can be improved by the usage of better techniques that allow the action to continue using alternate paths of execution.

The key to improved fault tolerance lies in the implementation of a mechanism for the system to monitor itself. The monitoring can be at several levels, discussed later, but the basic components of the monitoring system are *probes*.

4. System Monitoring and Probes

A distributed system supporting fault tolerance needs a system monitoring subsystem that effectively keeps track of the status of all the hardware components and software components (both active and passive). In this section we present a description of a probe-based monitoring system that can be coupled with the reconfiguration system to enhance the failure resilience of the system.

4.1. Implementing Probes in *Clouds*

Probes in *Clouds* are a form of emergency status enquiries, that can be sent from a process to an object or to another process. When a probe is sent to an object, the probe causes the invocation of a *probe-procedure* defined by default in the object. The probe procedure returns to the caller a status report of the object. This includes the status of the synchronization mechanisms, the actions currently executing in the object and other relevant information (Fig. 3).

Probes can also be sent to processes or actions. A process does not have to explicitly receive a probe. The probe causes a process thread of control to jump to the probe handler, irrespective of the current status of the process. (If the process is currently blocked, it executes the probe handler and returns to the blocked condition.) This is somewhat similar to the way an interrupt causes the CPU of a computer to call an interrupt handler. Thus probes to processes are conceptually similar to interrupts (or *signals* in the Unix operating system). The probe handler sends a message back to the originator of the probe, reporting the status condition of the process.

The probe handler in the process and the probe procedure in the object are pieces of code that are generated by default by the object compiler or the process compiler, or may be supplied by the programmer of the object or process.

The probe handler/procedures are scheduled and executed at higher priorities than the regular process scheduling priorities. Since the probes cause an immediate, asynchronous reply, and the probes are not suspended by synchronization mechanisms, the time taken

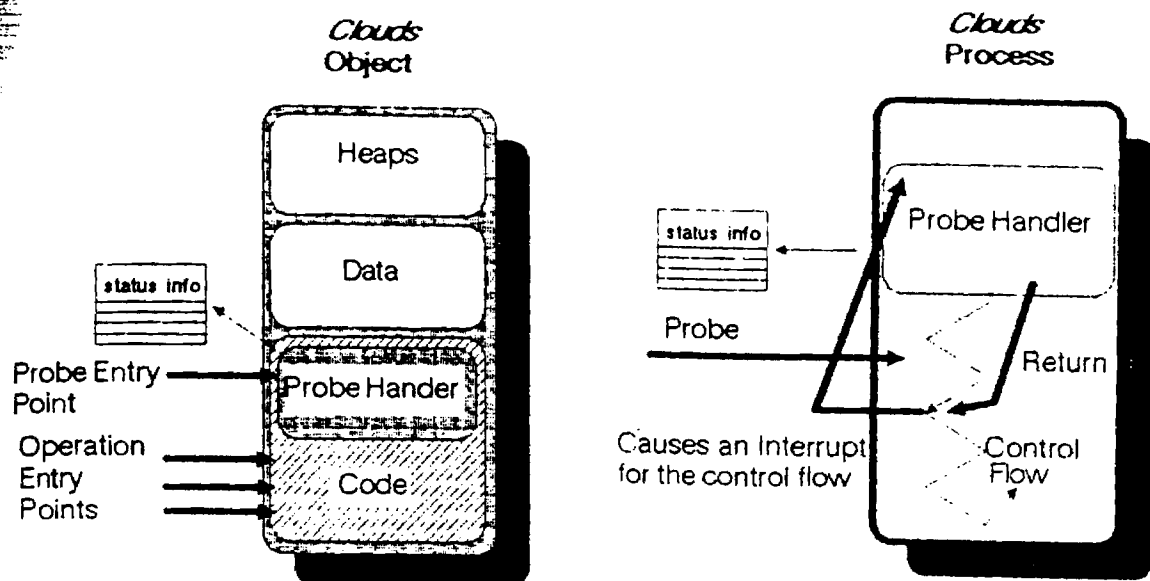


Fig. 3: Probe Handlers in Objects and Processes.

by the probe to return to the sender is not very dependent on unpredictable conditions, or heavy processing loads. Thus timeouts can be quite effectively used for receiving replies from probes. For example, if the system suspects a faulty network connection from system A to system B, a probe can be sent from system A to the network controller of system B. If the connection is indeed faulty, the probe will not return. System A can effectively determine this condition by timing out, retrying and timing out again. Thus using this scheme we can detect the difference between a dead machine and a slow one, with a high degree of accuracy.

4.2. The Primary/Backup Paradigm

The primary-backup paradigm can be used to enhance the fault tolerance of any crucial system process or action. For instance a critical action that needs to be executed to completion regardless of failures in hardware or communications can be handled as follows.

For each crucial action or process, associate a backup action (or process). The backup action has the same capabilities as the primary, but is dormant most of time. Periodically the backup process sends a probe to the primary. The primary probe handler sends back an *I am OK* status report. If the status report is not received in a certain amount of time the backup retries. If the retry fails, this implies the primary is dead, and the backup takes over as the primary and creates a new backup action.

The primary action also periodically checks on the backup. If the backup has failed, then the primary creates a new backup. For obvious reasons the primary and backup should be located at different sites. (Fig. 4.)

The drawback of this scheme is that if network partitioning occurs, we may get two primaries and two backups, as both the primary and the backup decides that the other process is dead. In *Clouds* however this produces no undesirable effects. Since the objects the two primaries have to access are the same, one (or both) of the actions will fail because of the unaccessability of some object due to the partition. So only one (if any) will run to completion.

This scheme handles *single stopping failures*. That is if any one process fails the system recovers. After the system recovers, another failure can be handled. But two failures occurring faster than the recovery time cannot be handled. The scheme can be modified to handle a multiple number of failures, but the complexity and overhead increases by a large factor. In *Clouds* it was deemed very unlikely that two failures will occur in such a short time and not worth the extra expense [McKe84a]

Clouds failure tolerance does not consider corruption or mass failure of storage media. Automatic recovery of storage media can be achieved by mirrored media, stable storage and so on. These methods are well understood, generally expensive and can be integrated into *Clouds*. We do not consider these techniques at this point.

4.3. System Health Monitoring using Probes

The primary/backup strategy is just one of the applications of the probe system. This scheme only utilizes the ability to send probes to processes. Since probes can be sent to passive objects as well they can be used in more situations. The more powerful application of probes is to build a system monitoring subsystem. System monitoring involves keeping track of the state of each system com-

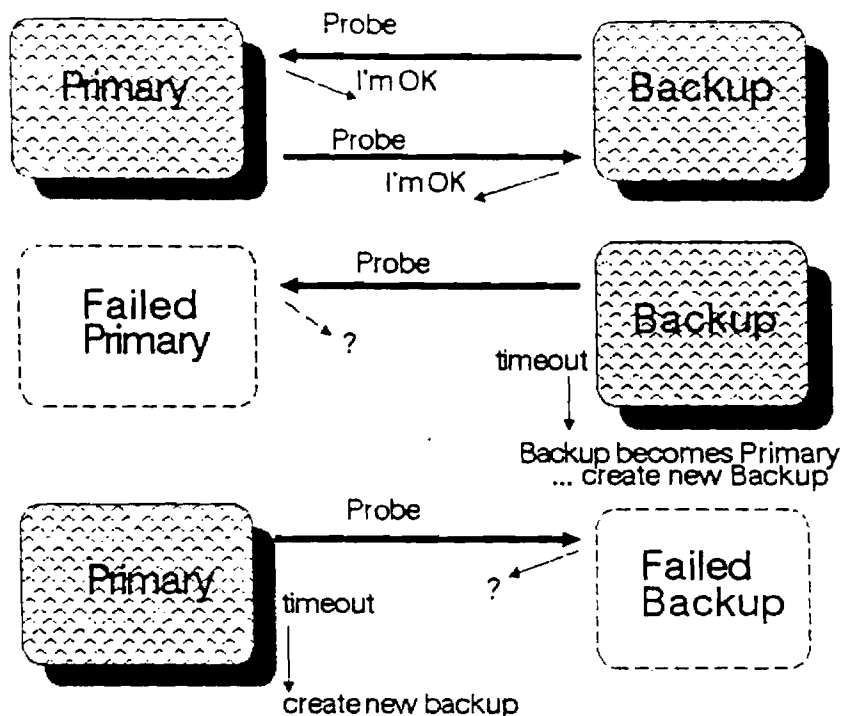


Fig. 4. The Primary - Backup Paradigm.

ponent that has any role to play in the overall function of the system. Passive components such as device interfaces, device control objects and so on, can be monitored by special monitoring processes that invoke diagnostic operations on these entities (or objects encapsulating the device drivers).

The system monitoring subsystem consists of a process (daemon) that runs at each site (monitor). The monitor has a list of components that it needs to keep track of. The list has a static part and a dynamic part. The static part contains capabilities to various critical system components (network drivers, disk drivers, schedulers, action management system and so on). The dynamic part consists of capabilities to user defined objects and actions that the user expressly records with the monitor, for tasks that require a high degree of fault tolerance.

The monitor at one site has a logical backup, that is a monitor at another site. The various monitors act as primaries for the site it runs on and doubles as a backup for a remote monitor. This allows the distributed system to detect site failures and network partitions. (Fig. 5.)

The monitor periodically probes all the components in its list. The status of these components are stored in a fully replicated database. This database has the same structure and properties as the database used to locate *Clouds* objects, i.e. it is highly available, but may not be consistent at all sites, or may contain out of date data. The inconsistency of the database does not cause major disruptions in service. The data in the database are used by various system services and the reconfiguration system.

Basically two kinds of failure information is available in the database. The first type is an entry that says a component X at site A is faulty or inaccessible. This is a local fault. The second type denotes that site A is unreachable or dead. This is a fault having global effect. (We are using an Ethernet as a local area network. For point to point communications, more data may be available, e.g. site A is unreachable from site B, but accessible via site C.)

The database updated by the monitoring system ties it to the *Clouds* reconfiguration system. The reconfiguration system and the interface to the monitor is discussed in the next section.

5. The Reconfiguration System

The reconfiguration system is responsible for maintaining normal system operations in the event of a detectable failure of one or more system components. *Clouds* recognizes two forms of reconfigurations, namely upwards and downwards. Downward reconfiguration occurs when some component fails, and upward occurs when some new or repaired component is brought on line. We will mainly discuss downward reconfiguration here, as failure is our major concern.

The reconfiguration system in *Clouds* handles downward reconfiguration by aborting all actions associated with the failed component. It then terminates all orphans

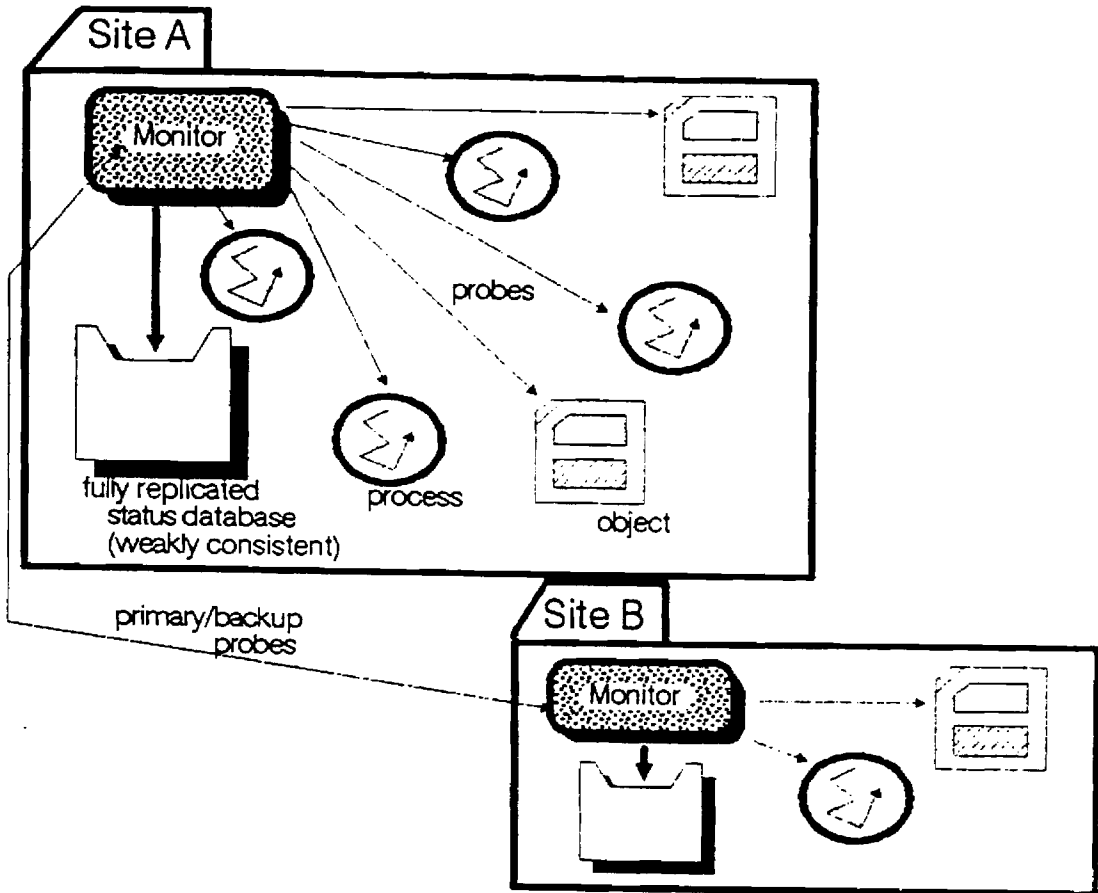


Fig 5: The structure of the monitoring system.

associated with the failed actions. The users who were affected by the failure are reassigned to functional sites and site search tables are updated to reflect the nonavailability of the malfunctioning site. The failure of a site also makes the objects stored at the site inaccessible. Access to these objects are restored by switching disk ports (if possible).

We divide the task of downward reconfiguration into two parts, namely *short term reconfiguration* and *long term reconfiguration*. Short term reconfiguration is used in the case of emergencies or large scale failures. This allows the system to substitute critical components and functionality and keep on running. Long term recovery uses monitoring systems to detect failure of non critical components, intermittent failures or bugs, anticipated failures and such, and tries to isolate faults.

Short term reconfiguration is handled as described above. When a failure is detected, all actions associated with this site and their orphans are aborted. If some of these actions were being run as fault tolerant actions, their backup counterparts would automatically take over. Access to objects that may become inaccessible due to the failure are restored, if possible, through alternate means.

Long term reconfiguration is more refined and adds stability to the system. This incorporates occasional analysis of the distributed (and replicated) directory maintained by the health monitoring system. Any faults or probable faults that may cause disruption are pinpointed and evasive measures taken. The evasive measures include changing the configuration of the system to avoid the usage of suspected hardware and creating alternate paths or alternate services to replace lost components.

Upward reconfiguration is a feature of *Clouds* that will be useful in most growing environments. This will allow additions of newer hardware, sites and network connections transparently. Upward reconfiguration is handled in two ways. If the new hardware is a device, then its drivers are added as objects, and any application that needs them can access them right away. If a site is added, it simply has to go on line. The search mechanisms will be able to find objects at the new site by its broadcast search mechanism, and the site will automatically enter the search hint files. The whole process is automatic and natural. No special method is needed for upward reconfiguration.

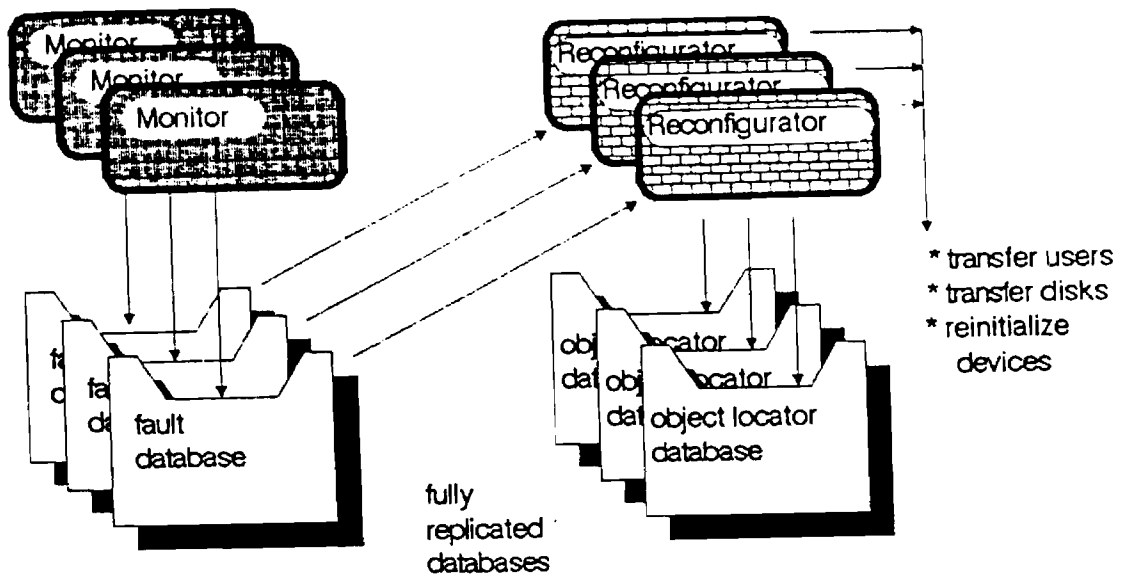


Fig. 6: Interfacing the Monitoring and Reconfiguration Systems

5.1. Interfacing Monitoring and Reconfiguration

The key to a successful reconfiguration system is the acquisition of the failure data. The monitoring system forms a useful source of data for the reconfiguration system.

The reconfiguration system is fully distributed, that is each site runs a reconfiguration daemon. The reconfiguration daemon periodically checks the failure database. (This can be somewhat improved. Details later). When a failure is detected the reconfiguration system works as follows:

If the failure is local, that is the reconfiguration process at site A finds that a component at site A is faulty, it updates the name service databases to this effect. That is this component is marked as unavailable for the system. The reconfiguration system then attempts to get the fault rectified. For example if the component is a device driver, it may delete and reload it from the template and reinitialize it. If the component is a disk, it will attempt to find an alternate path to the disk via a different site, and then transfer all the objects on the disk to the other site. Similar actions can be taken for an unresponsive user workstation.

If the failure is global, e.g. site A is unreachable, it will be detected at another site say B, where the backup was located. At this point the reconfiguration system at site B broadcasts update information to all sites so that no invocation request is sent to site A. It then attempts to transfer the disk connected to A to other sites having ports to A. All users on A are then floated off A to other functional sites.

The conceptual interfaces between the monitor, reconfiguration system, and the object locator database is

shown on Fig. 6. The handling of global failures (or site failures) is depicted in Fig. 7.

The periodic scanning of the fault database by the reconfiguration system may not get crash help soon enough. Quick notification of faults to the reconfiguration system by the monitoring system can be achieved by probes. After a critical fault is detected by the monitor, and added to the database, the monitor sends a probe to the reconfigurator, which immediately wakes up and does the needed repair.

The reconfiguration system is of course another fault tolerant system and should be under the supervision of the monitor. However failure of the reconfiguration system cannot be handled by the reconfiguration system and should be handled by the monitor. However failed monitors can be handled by the reconfigurator.

5.2. Long Term Reconfiguration

Most of the techniques described above are applicable for short term reconfigurations. Long term reconfiguration techniques present an interesting field of research that has not been adequately studied. The decisions that the long term reconfiguring system has to take are based on many factors such as importance of the system components, likelihood of failure of questionable components, the hardware system configuration and the availability and usefulness of alternate methods.

For example a lot of system components give rise to intermittent failures. Some of these problems disappear on their own, others work on retries, and then appear again. Especially on device interfaces, these problems can be often solved by reinitializing the device, or the driver.

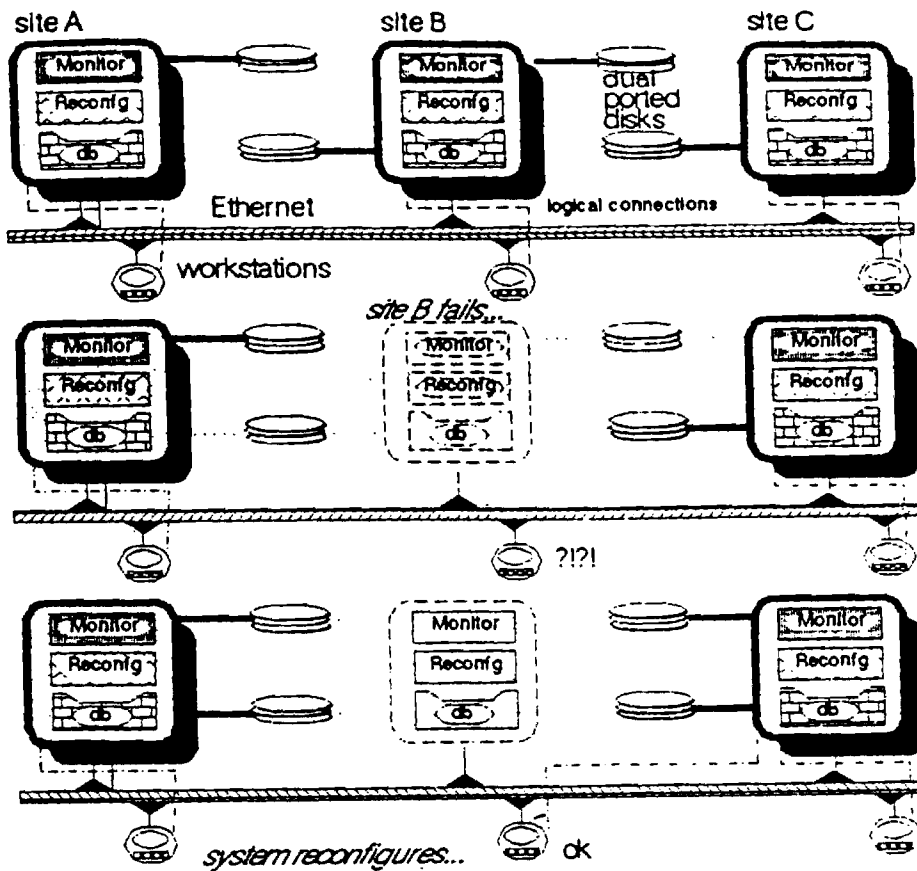


Fig. 7: Reconfiguration on Site Failure

The monitoring system can be used to keep track of the errors that occur on these components and maintain statistical information on the frequency of occurrences and severity. When errors rise above "normal" levels, the offending component can be rebooted if possible, or marked inaccessible. This can result in *early warnings* and help in the prevention of system crashes due to flaky components.

6. Debugging Support

As we have stated, the two basic software components of the system are processes and objects. Processes are active entities, executing on behalf of one task (or action). An object is a passive entity, that supports the execution of multiple tasks in its domain.

Several debugging techniques are under study, especially as debugging distributed systems use complex techniques to handle timing and concurrency aspects. Probes can be used to implement a simplistic yet highly effective debugging tool.

We treat processes and objects separately. A special *debug probe* sent to an object, causes the invocation of a interactive debugging routine, that resides in the object.

This routine allows the programmer to check the insides of an object *even when it is in motion*. This allows on the fly inspection of system objects without shutting the system down. Depending upon the object, repair on a running engine may be allowable.

A debug probe on a process causes the process to trap to a debug procedure. Similarly, the insides of the process can be checked. However, in this case, the process is not executing user code, and thus only asynchronous snapshots are possible. Some synchronicity can then be achieved by setting breakpoints, but the possibilities are not as powerful as the debugging of objects.

7. Conclusions

Clouds is a fault tolerant distributed system. The fault tolerance of *Clouds* is currently under implementation as an action-based backward recovery system. This type of fault tolerance leaves a lot to be desired. In this paper we have shown an easily implementable subsystem that effectively monitors the status of the distributed system, and actively helps the fault tolerant mechanisms to achieve

forward progress. The paper uses the useful probe mechanism to achieve this and proposes a simple implementation scheme for probes in the *Clouds* kernel, that can monitor both passive and active system components. Lastly, we present some techniques that uses probe based mechanisms for debugging objects and processes.

8. References

- [Al83] Allchin, J. E., *An Architecture for Reliable Decentralized Systems*, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983 (also released as technical report GIT-ICS-83/23)
- [AlBl83] Almes, G.T., Black, A.P., Lazowska, E.D., and Noe, J.D., *The Eden System: A Technical Review*, Technical Report 83-10-05, Department of Computer Science, Washington University, Oct 1983.
- [AlMc83] Allchin, J. E., and M. S. McKendry, *Synchronization and Recovery of Actions*, Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Montreal, August 1983
- [ChMi82] Chandy, K. M. and Mishra, J. A. *A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems*. Proceedings of the 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing.
- [DaLe85] Dasgupta P., LeBlanc R. J. and Spafford E. *The Clouds Project: Design and Implementation of a Fault Tolerant Distributed Operating System*. Technical Report, GIT-ICS-85/29. Georgia Tech, October 1985.
- [Jo79] Jones, A. K., *The Object Model: A Conceptual Tool for Structuring Software*, Operating Systems: An Advanced Course, Springer-Verlag, NY, 1979, pp. 7-16
- [Ke85] Kenley, G. *Design of an Action Management System for a Distributed Operating System*. Masters' Thesis, Georgia Institute of Technology, November 1985.
- [LiSc83] Liskov, B., and R. Scheifler, *Guardians and Actions: Linguistic Support for Robust, Distributed Programs*, ACM TOPLAS, Vol. 5, No. 3, July 1983
- [McAl82] McKendry M. S. and Allchin J. E. *Object-Based Synchronization and Recovery* Technical Report GIT-ICS-82/15, Georgia Instt. of Tech.
- [Mc84A] McKendry, M.S., *Clouds: A Fault-Tolerant Distributed Operating System*, Technical Report GIT-ICS-84/, Georgia Institute of Technology, 1984.
- [McHe85] McKendry, M. S. and Herlihy, M. *Time-Driven Orphan Detection*. Technical Report, Carnegie-Mellon University, CMU-CS-85-138, July 1985.
- [Mo81] Moss, J. E. *Nested Transactions: An Approach to Reliable Computing*. M.I.T. Technical Report MIT/LCR/TR-260.
- [PiSp85] Pitts, D. and Spafford E., *Notes on a Storage Manager for the Clouds Kernel*. Georgia Institute of Technology Technical Report, GIT-ICS-85/07.
- [Sp85] Spector, A.Z., Butcher, J., Daniels, D.S., Duchamp, D.J., Eppinger, J.L., Fineman, C.E., Heddaya, A., and Schwarz, P.M., *Support for Distributed Transaction in the TABS Prototype*, IEEE Transaction on Software Engineering Vol 11,6 (June 1985).
- [Sp86] Spafford E. *Kernel Structures for a Distributed Operating System*. Ph.D. Thesis, Georgia Institute of Tech., May 1986.