GEORGIA INSTITUTE OF TECHNOLOGY OFFICE OF CONTRACT ADMINISTRATION

PROJECT ADMINISTRATION DATA SHEET

[✓ X] ORIGINAL [ ] REVISION NO. ______

Project No. G-36-619 (R5815-0A0) GTRI/~~GIT~~x DATE 9 / 12 / 84

Project Director: Dr. Raymond E. Miller School/~~Lab~~ ICS

Sponsor: National Science Foundation

Type Agreement: Grant No. DCR-8405020

Award Period: From 9/1/84 To 2/28/86 (Performance) 5/31/86* (Reports)

Sponsor Amount:	This Change	Total to Date
Estimated:	$ 207,330	$ 207,330
Funded:	$ 207,330	$ 207,330

Cost Sharing Amount: $ 106,367 ** Cost Sharing No: G-36-561 (F5815-0A0)

Title: "Acquisition of Computer Research Equipment"

ADMINISTRATIVE DATA OCA Contact Lynn Boyd X4820

1) Sponsor Technical Contact:	2) Sponsor Admin/Contractual Matters:
Dr. John R. Lehmann	Myra B. Galinn
National Science Foundation	Grants Official
Computer Research Equipment	National Science Foundation
Computer Science Section	1800 G Street, N.W.
National Science Foundation	Washington, D.C. ~~20001~~ 20550
1800 G Street, N.W.	(202)357-9630
Washington, D.C. ~~20001~~ 20550	
(202)357-7349	

Defense Priority Rating: N/A Military Security Classification: N/A

(or) Company/Industrial Proprietary: N/A

RESTRICTIONS

See Attached NSF 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

COMMENTS:

*Includes usual 6 month unfunded flexbility period.

Equipment Grant.

**Cost-sharing may be reduced to actual requirements (not less than 25% of the costs).

OPAS approval for pre-award costs back to 8/17/84.

COPIES TO:

Project Director	Procurement/EES Supply Services	GTRI
Research Administrative Network	Research Security Services	Library
Research Property Management	Reports Coordinator (OCA)	Project File
Accounting	Research Communications (2)	Other I. Newton

FORM OCA 4:383

GEORGIA INSTITUTE OF TECHNOLOGY OFFICE OF CONTRACT ADMINISTRATION

SPONSORED PROJECT TERMINATION/CLOSEOUT SHEET

[illegible]

Date 7-1-87

Project No. G-36-619 School/~~Lab~~ ICS

Includes Subproject No.(s) N/A

Project Director(s) Dr. Raymond E. Miller GTRC / ~~GIT~~

Sponsor National Science Foundation

Title "Acquisition of Computer Research Equipment"

Effective Completion Date: 2/28/86 (Performance) 5/31/86 (Reports)

Grant/Contract Closeout Actions Remaining:

- [X] None
- [ ] Final Invoice or Final Fiscal Report
- [ ] Closing Documents
- [ ] Final Report of Inventions
- [ ] Govt. Property Inventory & Related Certificate
- [ ] Classified Material Certificate
- [ ] Other ____________________

Continues Project No. ____________ Continued by Project No. ____________

COPIES TO:

Project Director	Library
Research Administrative Network	GTRC
Research Property Management	~~Research Communications (2)~~
Accounting	Project File
Procurement/GTRI Supply Services	Other Duane H.
Research Security Services	Angela DuBose
[redacted]	Russ Embry
~~Legal Services~~	

FORM OCA 69.285

ATIONAL SCIENCE FOUNDATION
Washington, D.C. 20550

FINAL PROJECT REPORT
NSF FORM 98A

PLEASE READ INSTRUCTIONS ON REVERSE BEFORE COMPLETING

PART I—PROJECT IDENTIFICATION INFORMATION

. Institution and Address eorgia Institute of Technology chool of Information and Computer Science tlanta, GA 30332-0280	2. NSF Program Computer Science 4. Award Period From 9/1/84 To 2/28/86	3. NSF Award Number DCR-8405020 5. Cumulative Award Amount $ 207,330

. Project Title

PART II—SUMMARY OF COMPLETED PROJECT *(FOR PUBLIC USE)*

See attached report

PART III—TECHNICAL INFORMATION *(FOR PROGRAM MANAGEMENT USES)*

1. ITEM *(Check appropriate blocks)*	NONE	ATTACHED	PREVIOUSLY FURNISHED	TO BE FURNISHED SEPARATELY TO PROGRAM Check (✓)	Approx. Date
a. Abstracts of Theses					
b. Publication Citations					
c. Data on Scientific Collaborators					
d. Information on Inventions					
e. Technical Description of Project and Results	▨				
f. Other *(specify)*	▨				

2. Principal Investigator/Project Director Name *(Typed)* Raymond E. Miller	3. Principal Investigator/Project Director Signature	4. Date 5/22/87

SF Form 98A (3-83) Supersedes All Previous Editions — Form Approved OMB No. 3145-0058

Equipment for Reliable, Integrated, Distributed Computing.

Final Report

NSF Grant No. DRC-8405020
GIT Project Number G36-619

Principal Investigator:

Raymond E. Miller

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

1. Introduction

This report describes the use of the research equipment purchased with NSF equipment grant (DRC-8405020) for use in the *Clouds* project in the School of Information and Computer Science at Georgia Tech. The equipment was purchased in late 1984 and early 1985. Since then it has been put to significant use in software development and general research support relating to that project.

2. Equipment

The major equipment purchases under this grant have included three VAX 11/750 minicomputers, one IBM PC/AT and four IBM PC/XTs. Miscellaneous software, terminals and networking hardware were also purchased.

3. The Clouds Project

The *Clouds* project was initiated in 1982 to investigate techniques in building an effective reliable distributed operating systems and programming environments. The *Clouds* operating system was designed to consist of a distributed kernel that runs on bare hardware. The operating system support persistent objects at the native layer, location independence, distribution, and robust atomic transactions. The programming environment consists of programming languages, programming paradigms that exploit the distribution and reliability offered by the kernel, debuggers and object oriented tools.

The equipment purchased by under the grant has been heavily used in all phases of the project. The following is a brief description of the activity supported by the grant.

The *Clouds* project team currently consists of 4 faculty members and about 10 to 15 graduate students. They are being supported by the *Clouds* computing facility of the 3 VAX-750's, 5 PC's, 10 terminals and several Sun Workstations available as a part of the ICS computing facilities.

Of the 3 VAX-750's, one ("Stratus") is used to run Unix 4.2. This machine provides our stable work environment. The major usage of Stratus is in the *Clouds* kernel development and Aeolus compiler development. In the last two years the *Clouds* distributed kernel has been written and compiled on Stratus, and down-loaded to the other machines for testing. Other services provided to the users of Stratus include: CSNet mail, uucp mail, text formatting/typesetting, stable disk storage and access to all other machines on the ICS computing laboratory.

The other 2 VAX-750's ("Nimbus" and "Cirrus") is the *Clouds* testbed. Since *Clouds* is a native operating system, we need dedicated machines to test and run the kernel. Since *Clouds* is distributed, we need a minimum of two machines. Nimbus and Cirrus have been used for initial testing of *Clouds* since the middle of 1985, and has been running *Clouds* on a regular basis for the last eight months.

The IBM-PC's were targeted for use as workstations for *Clouds* users. A special purpose operating system was designed for use on the PC's for this purpose. However, this effort was abandoned when the Sun-3 workstations became available. The PC's purchased under the grant are being effectively used for graphics needs of the project (using the GEM desktop publishing software), spreadsheets for budgeting, local editing for papers and technical reports, and other purposes connected with program development, research publications and network access.

4. Progress

The *Clouds* project has developed a stable version of a distributed operating system kernel. The kernel is composed of a object management system, a storage management system and a communications system. In progress is the software development for the action management system and the user interfaces. The *Clouds* programming environment development has resulted in the development of the Aeolus language and compiler and allied software and paradigms suited for object oriented programming. The publication listed in the bibliography section of this report contain detailed description of the research and development that has taken place in the project.

Research in progress include development of monitoring techniques in distributed systems, fault tolerant action management, user interfaces, languages that support distributed object-oriented programming, robust programming paradigms, object programmers toolkit (Pandora) and distributed systems management.

The equipment grant allowed the *Clouds* project to to reach towards the goals that were envisioned in 1983 (and which survived several revisions), and reach a point where the project has received recognition and credibility amongst researchers in this area.

The *Clouds* project has been funded from several sources including NSF, NASA and the Dept. of Defence. Most of this support has been towards personnel. The NSF equipment grant has been very important in providing the right set of equipment for the project to achieve progress. The equipment has also been the key resource that has enabled the project to receive personnel funding from various sources.

The progress of the *Clouds* project and results obtained by the project has been useful in synthesizing the Georgia Tech proposal to the NSF-CER program for 1986. (The proposal is pending). The *Clouds* environment was proposed as a base layer for the large general purpose computing environment for computer science research. The *Clouds* project received good reviews in the CER site visit and the major contributing factor to the success of the *Clouds* approach is that we have been able to demonstrate the feasibility of a distributed operating system built using our approach through the actual building of the system. This would never have been possible without the equipment we have been using for the last few years.

5. Conclusions

The equipment purchased under the NSF grant DRC-8405020, has allowed the *Clouds* project to move from a set of ideas to a demonstrated system that has produced some significant research results in the area of distributed operating systems and distributed environments.

The research environment created by the equipment has stimulated the researchers in the project to produce ideas, papers, technical reports, and above all, software that works as designed.

Bibliography of Work Supported by this Grant

1. Ahamad, M., M. H. Ammar, J. Bernabeu, and M. Y. A. Khalidi, "A Multicast Scheme for Locating Objects in a Distributed Operating System," Technical Report GIT-ICS-87/01, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, January 1987.

2. Ahamad, M., P. Dasgupta, R. J. LeBlanc, and C. T. Wilkes, "Fault-Tolerant Computing in Object Based Distributed Operating Systems," *Proceedings of the Sixth Symposium on Reliability in Distributed Software and Database Systems*, pp. 115-125, IEEE Computer Society, Williamsburg, VA, March 1987.

3. Ahamad, M. and P. Dasgupta, "Parallel Execution Threads: An Approach to Fault-Tolerant Actions," Technical Report GIT-ICS-87/16, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, March 1987.

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

5. Dasgupta, P. 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.

6. Dasgupta, P., "A Probe-Based Monitoring Scheme for an Object-Oriented Distributed Operating System," *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pp. 57-66, ACM SIGPLAN, Portland, OR, Sept. 1986. Also available as Technical Report GIT-ICS-86/05

7. Kenley, G. G., "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

8. LeBlanc, R. J. 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

9. McKendry, M. S., "Ordering Actions for Visibility," *Transactions on Software Engineering*, vol. 11, no. 6, IEEE, June 1985. Also released as technical report GIT-ICS-84/05

10. Pitts, D. V. 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.

11. Pitts, D. V., "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

12. Pitts, D. V. and P. Dasgupta, "Object Memory and Storage Management in the Clouds Kernel," Technical Report GIT-ICS-87/15, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, March 1987.

13. 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, GA, 1984.

14. Spafford, E. H., "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

15. Spafford, Eugene H., "Object Operation Invocation in Clouds," Technical Report GIT-ICS-87/14, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, February 1987.

16. Wilkes, C. T., "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

17. Wilkes, C. T. and R. J. LeBlanc, "Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System," *Proceedings of the 1986 International Conference on Computer Languages*, pp. 107-122, IEEE Computer Society, Miami, FL, October 1986. Also available as Technical Report GIT-ICS-86/12

The Clouds Distributed Operating System.
Functional Description,
Implementation Details
and
Related Work.

Partha Dasgupta, Richard LeBlanc & William Appelbe.

Submitted to

The 7th International Conference on
Distributed Computing Systems.

(Distributed Operating Systems Group)

Abstract

Clouds is an operating system targeted to be a prototype of a novel class of distributed operating systems providing the integration, reliability and structure that makes a distributed system generally 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 decentralized yet integrated control.

The design criteria for *Clouds* include integration of resources through location transparency, support for robust transaction processing as well as advanced support for achieving fault tolerance, provisions for dynamic reconfiguration and an object based system architecture. The implementation has been tailored to be simple, efficient and usable.

The system structuring paradigm chosen after substantial research for the *Clouds* operating system is an object/process/action model. All instances of services, programs and data in *Clouds* are objects. Processing can be done within or outside the atomic action construct. The concept of persistent object does away with the need for file systems, and replaces it with a more powerful object system. Concurrency control and recovery is handled within the objects. In this paper, we provide a functional description of the system, planned enhancements, some implementational details, and discussion of related work.

Contact Address:
Partha Dasgupta
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

Phone: +1 (404) 894 2572

Electronic Address:
CSNet: partha@GaTech.CSNet
Arpa: partha%GaTech.CSNet @ CSNet-Relay.ARPA
UUCP: ...!{akgua,allegra,amd,hplabs,ihnp4,masscomp,ut-ngp}!gatech!partha

The Clouds Distributed Operating System[†]. *Functional Description, Implementation Details and Related Work.*

Partha Dasgupta, Richard LeBlanc & William Appelbe.
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

1. Introduction

Clouds is an operating system targeted to be a prototype for a class of distributed operating systems providing the integration, reliability and structure that makes a distributed computing system generally 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 structure of *Clouds* promotes transparency, support for advanced programming paradigms, and decentralized yet integrated control. The major design objectives for *Clouds* are:

- Integration of resources through cooperation and location transparency.
- Support for robust transaction processing, and the ability to achieve fault tolerance.
- Efficient design and implementation.
- Simple and uniform interfaces for distributed processing.

The system structuring paradigm chosen after substantial research for the *Clouds* operating system is an object/process/action model. All instances of services, programs and data in *Clouds* are objects. Processing can be done either by atomic actions or by processes that execute outside the constraints of atomicity [Al83, DaLe85, McAl83]. In this paper, we provide a functional description of the system (sections 2 to 8), planned enhancements (section 10), some implementational details (section 11), and discussion of related work (section 12). Currently, an operational distributed prototype of the *Clouds* kernel has been implemented, and is in experimental use by the developers. We plan to put the kernel in more widespread use over the next few months (section 9).

2. Objects

All data, programs, devices and resources on *Clouds* are objects. The only entities that are not objects are processes and actions. A *Clouds* object, at the lowest level of

[†]This research has been partially funded by NASA (grant NAG-1-430) and NSF (grant DCS-83-16590).

conception, is a virtual address space. Unlike conventional virtual address spaces, a *Clouds* object is neither tied to any process nor is volatile. A *Clouds* object exists forever (like files) unless explicitly deleted. As will be obvious in the following description of objects, *Clouds* objects are somewhat 'heavyweight' rather that 'lightweight' objects provided by Smalltalk for example.

Every *Clouds* object is named. The name of an object, also known as the 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.

Since an object consists of a named address space (and its contents), it is completely passive. Unlike those in some object based systems, a *Clouds* object is not associated with any server process. Processes are allowed to execute within the context of objects. A process executes in an object by entering it through one of several entry points, and after the execution is complete the process leaves the object. Several processes can simultaneously enter an object and execute concurrently.

Objects have structure. They contain, minimally, a code segment, a data segment and a heap for local storage allocation. Processes 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 processes).

Clouds objects are user-defined or system-defined. Most objects are user-defined. Some examples of system defined objects are device drivers, name-service handlers, and the *Clouds* kernel[†] itself. 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. The *Clouds* object structure is depicted in Figure 1. Since a conventional file is a special case of a *Clouds* object, the *Clouds* object system does away with the need for a file system. This is discussed in further detail in section 4.

Each object in the *Clouds* system is 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

[†]Although the specification defines the kernel to be an object, the implementation treats it as a special case (or pseudo-object) for efficiency reasons.

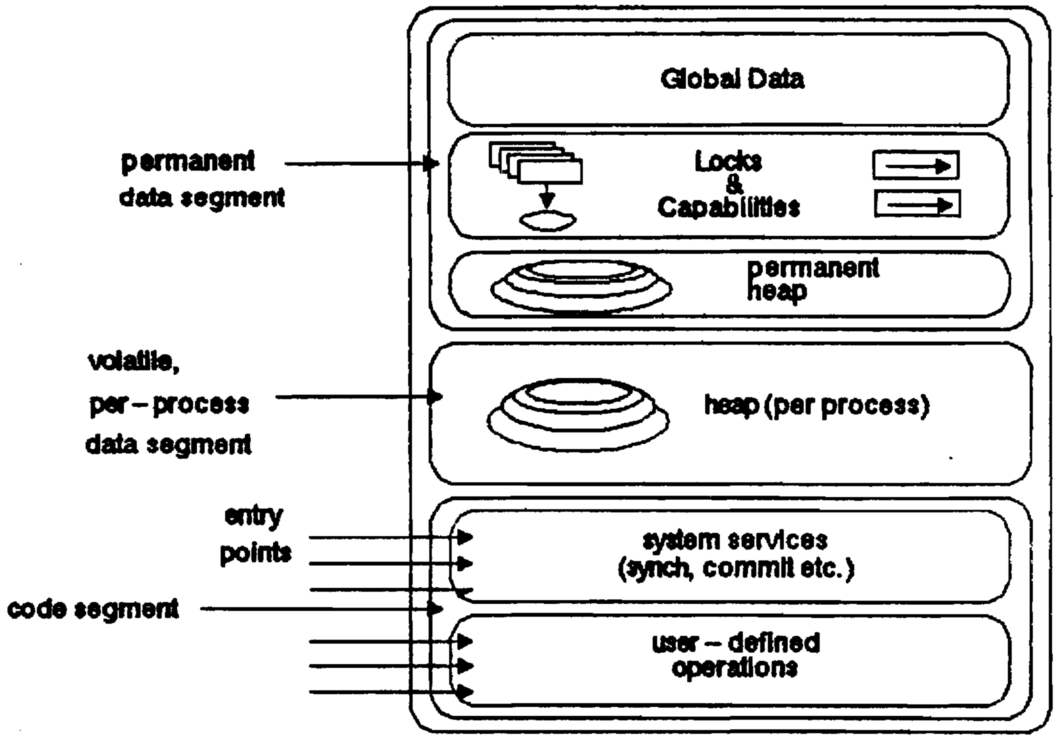

Figure 1: Clouds Object Structure.

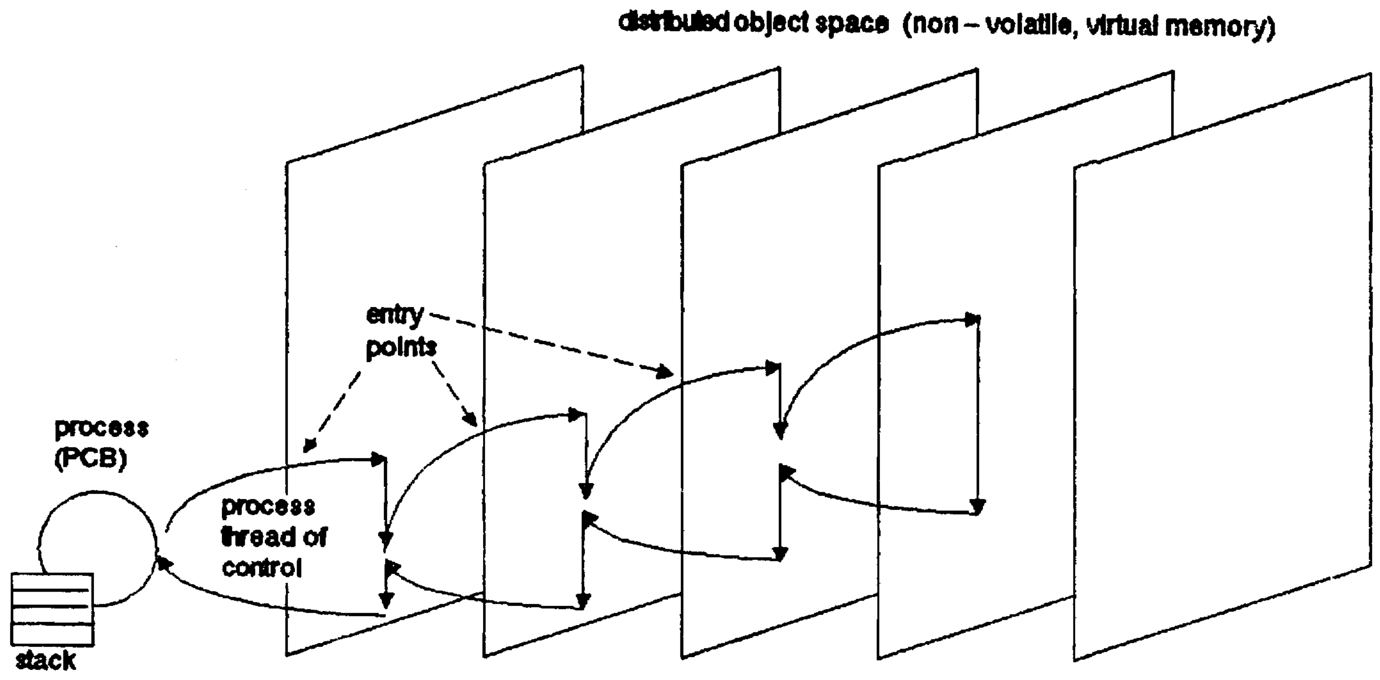
distributed object space (non-volatile, virtual memory)
entry points
process (PCB)
process thread of control
stack

Figure 2: Object Memory in Clouds.

template. The templates, the template-template and all the instances thereof, are regular *Clouds* objects, and as discussed earlier, they exist when created, and live unless explicitly deleted.

3. Processes

The only form of activity in the *Clouds* system is the process. *Clouds* processes are lightweight workers. A process is composed of a process control block (PCB) and and a virtual space containing the stack. Thus, a process can be viewed as a program counter, stack pointer and stack. Upon creation a process starts up at an entry point of an object. As the process 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 process executes this, it temporarily leaves the caller object and enters the called object, and commences execution there. The process returns to the caller object after the execution in the called object terminates. The calls to the entry point of objects are called *object invocation*. Object invocations can be nested. The code that is accessible by each entry point is known as an *operation* of the object.

A process executes by processing operations defined inside many objects. Unlike processes in conventional operating systems, the process often cross boundaries of virtual address spaces. Addressing in an address space is however limited to that address space, and thus the process cannot access any data outside an address space. Control transfer between address spaces occurs though object invocation, and data transfer between address spaces occurs through parameters to object invocation (which may be capabilities for other objects).

When a process 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. Since the address spaces of the two objects are disjoint, all arguments are passed by value. This argument passing mechanism is identical to copy-in copy-out semantics of parameter passing supported by many programming languages.

4. The Object/Process Paradigm

The structure created by a system composed of objects and processes has several interesting properties. First, all 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 concept of machine boundaries. At the system level local invocations and remote invocations (also known as remote procedure calls or RPC) are differentiated, however this is transparent to the user or systems programmer.

Since every entity is an object and objects are permanent, there is no need for a file system. A conventional file is a special case of an object, an object with a read, a write, a seek and some other file operations defined in it to transport data in and out of the object through parameters provided to the calls.

Though we can simulate files by using objects of type *file*, the need for files disappear in most situations. Programs, do not need to store data in files, they can keep the data in the data space in each object. Since the data space does not disappear when the controlling process terminates. The need for user-level naming of files transforms to the need for user-level naming for objects.

The *Clouds* operating system does not provide any support for I/O operations, except for terminal I/O. (Terminal I/O is achieved by invoking the read and write operations on a terminal object, dispensing with most concepts about I/O streams).

Just as I/O is eliminated, so is the need for messages. Processes do not communicate through messages. There are no ports. This allows a simplified system management strategy as the system does not have to maintain linkage information between processes 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 programmers view of the computing environment created by *Clouds* is apparent. It is a simple world of named address spaces (or objects). These object live in computing systems on a LAN, but the machine boundaries are made transparent, creating a unified object space. Activity is provided by processes 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* and is depicted in Figure 2.

This view of a distributed system, does have some pitfalls. Processes aborting due to errors will leave permanent faulty data in objects they modified. Failure of computers will result in similar mishaps. Multiple processes 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 the atomic actions paradigm, discussed below.

5. Actions

Actions are units of work that are defined by the programmer to be atomic. The work done by an action either gets done in its entirety or does not happen at all. Failures, errors, and aborts thus do not leave a trace on the data stored in the objects [Ke86, Mc85].

An action is an abstraction. It is neither an object nor a process. It is a high level concept that exists as information in the action management system. An action starts as an invocation of an entry point in a process. This is a top-level action, and is started by executing a 'start-action' system call to the *Clouds* kernel, with the target object and operation as parameters. Then, a process is created, which executes on behalf of this action, and invokes the specified object. Anything this process does, until the action commits or aborts, is in the context of the action. If the process creates more processes, invokes other objects and creates subordinate actions, these too are part of the same top-level action.

All the activity of the set of processes (or one process) in the context of an action, consists of *touching* objects. A object is considered *touched* by an action if a process executing on behalf of the action executes one or more object invocations on the object. A touched object is not necessarily modified. All objects modified by an action exist in a volatile form that may be different from their permanent representations.

When an action terminates, all the objects touched by the action are *committed*. Commitment of an object is achieved by updating its permanent representation by replacing any data that have been modified by this action. Since the updates by an uncommitted action are never made permanent, an aborted action is rolled back by default.

Though the updates by an uncommitted action are not written on permanent (secondary) storage, the updates of an uncommitted action may be seen by another uncommitted action accessing the same object, depending upon the synchronization method used by the object. We distinguish between two kinds of atomicities of actions, namely *failure atomicity* and *view atomicity*. Failure atomicity dictates that either the updates performed by an action are made permanent after the action runs to completion, or nothing occurs. View atomicity dictates that the action is insulated from seeing any results from other concurrently executing actions. *Clouds* can provides failure atomicity and, if needed, view atomicity. Note that not providing view atomicity can lead to errors (an action A makes updates based on some results of an uncommitted B action and A commits while B aborts). The differences between the atomicity requirements and the rationale for providing failure atomicity will be clearer after the discussion of synchronization methods.

5.1. Nested Actions

Actions, as units of work, are too large grained for many applications, especially in an large distributed environment where failures are relatively common. Any error or failure during the execution of an action requires that the entire action be aborted. An action often needs the ability to recover from errors or failures. Finer grained atomicity and failure recovery capabilities are provided by nested actions.

A top-level action is the conventional action. The-top level action can delegate sub-tasks to subordinated actions or sub-actions, which in turn can spawn sub-actions, giving rise to a tree of nested actions. A child action executes in the context of its parent, but the failure of the child does not imply the failure of the parent, the parent may choose to retry the sub-task or respond to the failure in some other way. The commit of a sub-action is conditional upon the commit of the parent, and by transition, the commit of all nested actions are conditional upon the commit of the top-level action. Thus a top-level action makes the final commit decision, based on the commit and abort status of all the nested action it gave rise to.

The nested action semantics of *Clouds* is identical to the semantics defined by Moss in [Mo81]. Nested actions thus provide a action programmer with failure containment firewalls and the ability to try alternate methods to make progress.

6. Synchronization

The synchronization scheme decides how (if at all) concurrent processes execute in the same object. The synchronization scheme used also dictates whether action using the object are view atomic or not. Both the synchronization techniques used and the recovery techniques used affect the semantics of action atomicity. We discuss synchronization in this section. Recovery will be discussed in the next section and the effects of both on actions will be briefly considered [McAlMc82].

Clouds provides two types of synchronization: *custom* and *automatic*. Custom synchronization allows the programmer of an object to define and implement the synchronization rules. For this purpose, the programmer has access to locks and semaphores that can be defined and used inside the object. For example, setting a lock on a variable when entering an operation and releasing it upon exit causes processes that execute this operation to run in mutual exclusion. The programmer can thus customize the synchronization scheme to the needs of the object.

Though custom synchronization can be correct and useful for many applications, it is possible to cause non-serializable execution in custom synchronization schemes. Allowing various unconventional schemes is the power of custom synchronization. However in cases where serializability is necessary, the programmer need not implement any synchronization; automatic synchronization is available for this purpose.

With automatic synchronization, each entry point in an object is marked as a read entry or a write entry. When an action touches an object for the first time, a read or write lock is obtained (as appropriate) on the entire object. Conversions from a read lock to a write lock is allowed. Locks are held until the action commits, implementing a two-phase locking protocol and guaranteeing serializable execution of the action with respect to all data touched by the action (provided all objects it touched were using automatic synchronization). This scheme provides view atomicity of actions.

7. Recovery

Recovery is managed by shadowing, providing failure atomicity for actions. Objects are classified as recoverable or non-recoverable. Non-recoverable objects are somewhat cheaper to handle and can be used by non-critical system tasks, but usage of non-recoverable objects by actions can lead to lapse of consistency. Note that all Unix files are identical to a non-recoverable *Clouds* object of a specific type (unstructured file).

When an action invokes an operation in a recoverable object, a *shadow* version and a *core* version of the object is created. The shadow version is the original permanent version, and the core version is the possibly updated version. If several actions invoke an object in parallel, there is still only ONE shadow and ONE core version. If the synchronization is not automatic, there are possibilities that one uncommitted action will see updates from another uncommitted action, violating the view atomicity requirements (if any). But this is left to the programmer who chose the synchronization strategy.

Every recoverable object has two default entry points called *pre-commit* and *commit*. When the pre-commit entry point is invoked, the object flushes all the updated data in the core version to stable storage, and the commit operation copies the updates to the shadow version and makes the shadow version the permanent version. These entry points can be used by any 2-phase commit protocol.

Like synchronization, recovery comes in two flavors, namely custom and automatic. When an action is run with automatic synchronization, the action management keeps track of all the objects the action touched. When the action terminates successfully, the action management system creates a commit co-ordinator process, that uses the pre-commit and commit entry points of all the objects touched by the action to perform an atomic commit, using the two-phase commit protocol.

Custom recovery is nearly identical, except that the programmer has the ability to redefine the default pre-commit and commit routines in the objects; the user defined routines will be used by the action manager at commit time. The user also has access to the commit routines during normal execution and thus can perform intermediate check points, partial commits and customized features like flushing only certain pages of the object.

Automatic recovery and automatic synchronization guarantee serializability, failure atomicity and view atomicity. Automatic recovery and custom synchronization guarantees failure atomicity and allows the user to use some concurrency control semantically consistent with the application. Custom recovery and synchronization allows the programmed full control of the execution strategy, and the system does not guarantee anything other than the synchronization and recovery support.

8. 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 provides linguistic support for programming *Clouds* objects and allow the composition of objects from sub-objects. Aeolus provides access to the synchronization features (both custom and automatic) and the recovery features of *Clouds*. Though the *Clouds* programmer is not tied to Aeolus, the language is most suited for systems programming as it has been tailored to match the kernel features [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.

9. Current Status

The *Clouds* distributed kernel is operational. The implementational details are discussed in section 11.

Clouds runs on a set of VAX-11 computers. *Clouds* is a native operating system, that is it runs on bare hardware without any support from some other operating system. Currently we support the object memory, capability based invocations, location transparency, custom synchronization and recovery though a locking facility and a shadow page recovery scheme.

Application programs have been tested using C and Aeolus as the source languages. The Aeolus compiler and linker runs under Unix and the object code is transferred into

Clouds objects though transfer utilities. The Aeolus runtime support system provides argument passing between *Clouds* objects and some terminal I/O support.

The user interfaces to *Clouds* is via Unix. The *Clouds* utilities and application programs link to Unix workstations (Sun-3/50's) over the Ethernet, and allows *Clouds* programmers access to full Unix user support, and thus the transition from Unix to *Clouds* is made easy for those used to Unix. Unix programs are able to invoke *Clouds* objects and thus use *Clouds* facilities.

The action management system is under implementation, and we hope to provide automatic recovery support in about three to four months. *Clouds* is going to be used in some graduate courses for distributed computing projects, and we will get additional user input when we start widespread use of the *Clouds* system amongst the local user community.

10. Enhancements and Planned Features

The above description of *Clouds* documents the basic features of the distributed kernel for *Clouds*. Presently the following enhancement, applications and features are at various stages of design, implementation and planning.

- An object naming scheme is being developed that creates a hierarchical user naming strategy (like Unix) that is also highly available and robust (through replicated directories).
- Unix and *Clouds* will be inter-operable providing Unix programmers and user with access to *Clouds* features and *Clouds* programmers to use Unix services. Unix machines will be able to execute remote procedure calls to *Clouds* object thus gaining access to all the functionality that *Clouds* provides. In fact the user interface to *Clouds* will be through Unix shells and tools. Similarly *Clouds* applications will make use of the wide variety of programming support tools that are supported by Unix through a mechanisms that provides Unix service for *Clouds* computations. In addition, *Clouds* services will be directly accessible through *Clouds* libraries for other programming languages, such as C++ and ADA.
- As mentioned earlier, mechanisms for providing object-oriented programming methodology will be provided at the linguistic level, with enhancements in the kernel that will provide performance advantages (such as sharing of code in the classes with its instances).
- Debugging support at the object level, process level and the invocation level will be provided. Techniques that allow the programmer to get a comprehensive view of the distributed and concurrent execution environment are under development.

- A probe system that can track object and process status in the system can provide information about failures, loading, deadlocks and software problems is being developed. This will be used to develop a distributed system monitoring system that will help in reconfiguration on failure and aid in providing fault tolerance. The probe system will also be useful in distributed object level debugging [Da86].
- A distributed database that utilizes the object structure of *Clouds* for elegance and the synchronization and recovery support for concurrency control and reliability is being developed [DaMo86].
- *Clouds* has been designed as a base layer for fault tolerance computing. The systems that will provide fault tolerance and guarantee progress of computation and system response in face of partial system failures are being developed. The techniques include replicated objects, multi-threaded actions, the coupling of the reconfiguration systems and monitoring systems, and usage of dual-ported storage devices.

11. 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. Non-homogeneous 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 synchronization and recovery systems should be efficient.
- Since one of the primary aims of *Clouds* is to provide the substrate for reliable, fault tolerant computing, the base system design should be tolerant to failures and provide adequate support for implementing fault tolerance.
- The system design should be simple to comprehend and implement.

11.1. Hardware Configuration

The hardware being used for implementing the prototype *Clouds* system is commonplace: three VAX-11/750's connected by an Ethernet. The disk units are dual ported, allowing access to the units from two machines, which provides the ability to remount the data from one machine to the other in case of site failures thus increasing availability.

The user interface is not through terminals, but over the Ethernet from Unix mainframes or workstations. This allows easy (software based) reassignment of users in case of site failures.

11.2. Software Configuration

The *Clouds* kernel is a native kernel running on bare hardware. The structure of the distributed system and the per site kernels are shown in figures 3 and 4. 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. Aeolus has been used as the implementation language for *Clouds* utilities.

11.3. Kernel Structure

The kernel is a replicated resident kernel, replicated at all the sites. Logically, the kernel is distributed over several sites and the machine boundaries are invisible. This is achieved by the communication system that provides the low level messaging interface between the replicated kernels. The system control however is completely decentralized, so that failure of individual kernels do not affect the rest of the system [Sp86].

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 most of the functionality of conventional operating systems (such as Unix), building *Clouds* on top of a Unix like kernel would have unacceptable deficiencies such as unwieldy implementation and poor performance. Secondly, the paradigms and the support used in *Clouds* is 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*. Some of the *negative* aspects of using standard Unix implementations as the base layer would be:

- Unix processes are heavyweight processes: thus process creation and RPC would be expensive.
- A Unix process is tied to one address space. Making a process cross address spaces would involve simulating it through multiple processes and the Unix IPC mechanism, which would involve multiple context switches and other message layer overheads.
- Only one process can execute in an address space, providing serious limitations to intra object concurrency. There are methods that get around this problem, but they are generally complex to implement, unreliable and require substantial overhead.

To avoid these problems the *Clouds* kernel is designed to support the *Clouds* functions on native hardware and all the performance critical support is implemented at the lowest level in the kernel.

11.4. Object Naming and Invocation

The two basic activities inside the *Clouds* kernel are system call handling and object invocations. System call handling is done locally, as in any operating system. The system

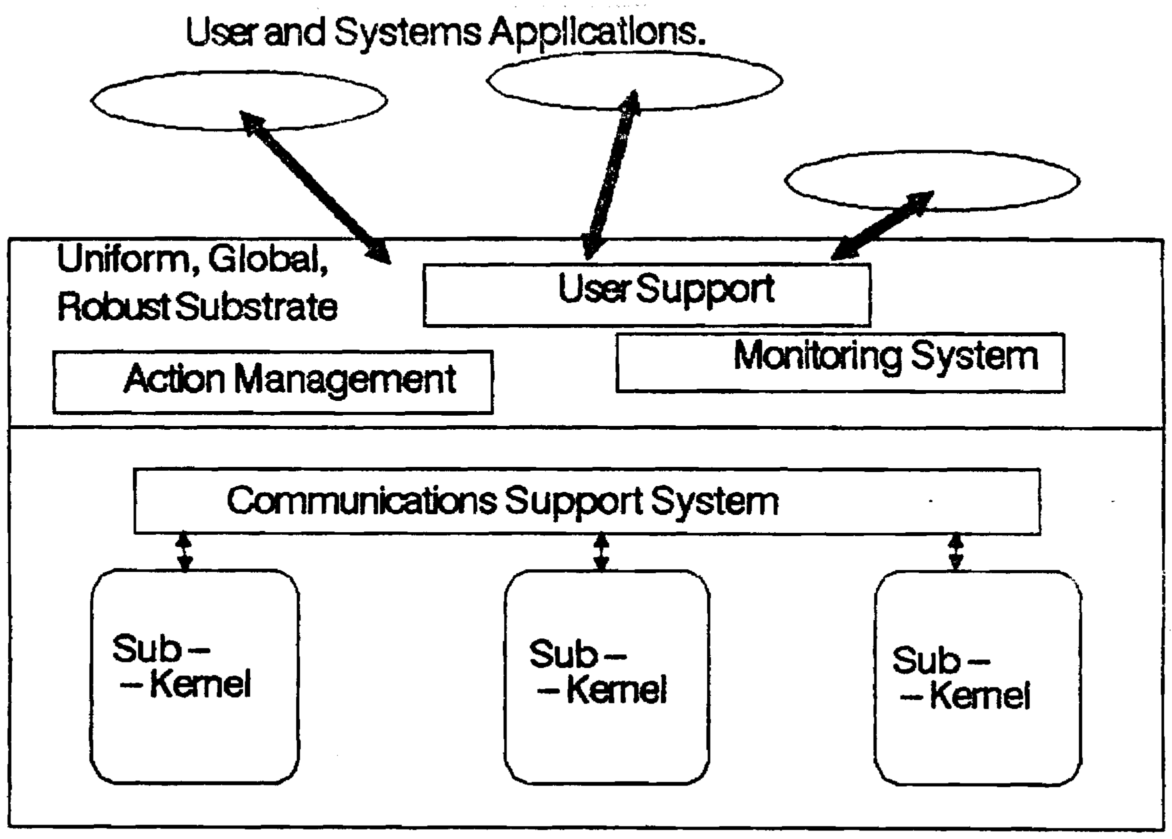
User and Systems Applications.
Uniform, Global, Robust Substrate
User Support
Monitoring System
Action Management
Communications Support System
Sub--Kernel
Sub--Kernel
Sub--Kernel

Figure 3: The Software Structure of the Clouds Distributed System.

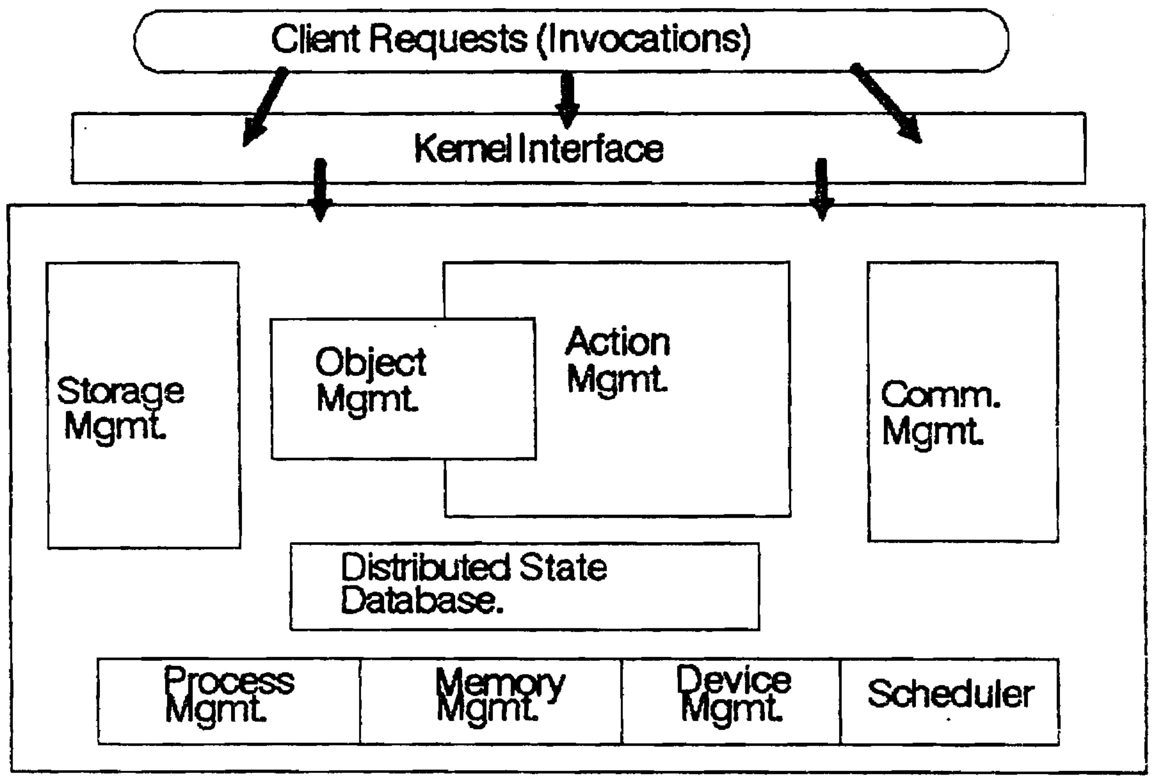

Figure 4: The Structure of the Clouds Kernel (per site).

calls supported by the *Clouds* 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 processes. The attributes that object invocation must 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. Also, moving objects between sites, reassigning disk units and so on should be simple.

Location independence is achieved through a capability based naming system. Availability is obtained through decentralization of directory information and a unique search-and-invoke strategy. Speed is achieved by implementing the invocation handlers at the lowest level of the kernel, on the native machine.

When a process invokes an object, it first places the arguments on the stack and executes an *invoke* system call, with the called object capability as the parameter. The capability of the object is unique systemwide, but has no site information. The kernel searches the local object directory to determine if the object is available locally. If it is, then the process address space is switched and the process starts executing in the object that it invoked. (This is achieved by changing the P0 region of the VAX address space by updating the P0 page table registers. The stack of the process is in P1 region, and this space remains the same.)

If the object does not exist locally, the kernel broadcasts a search-and-invoke request. All participating kernels then attempt to locate the object. The successful kernel dispatches a slave process, which copies the arguments from the invoke request to its stack and performs a local invocation on the object. Upon termination, the arguments are send back to the invocation requester, which causes the invocation request to return.

Hash tables, caches, and hint databases are used to add speed both the local searches for objects as well as avoiding the need for all sites to search for objects at each broadcast search-and-invoke request. A special hashing scheme that uses multicasting has been developed that reduces the search overhead by a large margin [AhAm87].

11.5. Storage Mangement

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 [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.

11.6. Action Management

The storage management implements the virtual memory system and the commit protocols, providing the mechanisms for handling the object storage needs. The policies of the action management are not implemented in the storage manager, but rather in the action management system. The action management system implements nested actions for the *Clouds* system by keeping track of the objects touched by an action as well as the success and failure of each action and its subactions [Ke86].

The action manager primarily keeps track of information regarding actions. The action manager is distributed, with the manager at each site keeping information about each action that was started as a top level action at that site. Although an action can span several sites, the action commit is coordinated through the action manager at the site where the action started. As is apparent if the site starting the action fails, the action is doomed to abort anyway, and hence the failure of the coordinating action manager does not matter in this case.

As discussed previously, when an action terminates, the coordinating manager invokes commit operations on all touched objects, in order to make all updates by the action permanent in an atomic step.

12. Comparisons with Related Systems

Clouds is one of the several research projects that are actively building distributed object based environments. There are similarities and differences between all the approaches, and the area of distributed operating systems are in general not mature enough to conclusively argue the superiority of one approach over the other. In the following paragraphs we document some of the major differences between *Clouds* and some of the better know projects in distributed systems.

One of the major difference between *Clouds* and most 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. In addition, no attempt has been made to build a UNIX interface 'on top of' *Clouds*. *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, etc.)

12.1. Argus

Argus is a system developed at MIT, that supports the Argus programming language. 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 any operation on atomic data types by actions are updated atomically when the action terminates [WeLi83, LiSc83].

Some of the similarities between Argus and *Clouds* are the semantics of nested actions. Both use the nested action semantics and locking semantics described by Moss. This includes conditional commit by subactions and lock inheritance by subactions from the parents as well as lock inheritance by the parents from a committed child. Also the guardians and handlers in Argus have somewhat more than cosmetic similarities to objects in *Clouds*.

The differences include the implementation strategies, programming support and reliability. 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 Aeolus 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. Unlike Argus, *Clouds* is designed to form the base layer for fault tolerant computing.

12.2. Eden

Eden is a object based distributed operating 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. The messages use the Berkeley Unix IPC mechanism [Alm83, AlBl83, NoPr85].

†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.

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 and it reads in the passive representation into its virtual space and then performs the required operation. The activation of passive objects is an expensive operation. Also concurrent invocations of objects are difficult and is handled through multithreaded processes or coroutines.

The active object paradigm and the Unix based implementation are the major differences between Eden and *Clouds*. This is also the reason for the performance problems in Eden. Eden also provides support for transaction and replication objects (called Replects). The transaction support and replication was added after the basic Eden system was designed and have some limitations due to manner Unix handles disk I/O. Eden was not designed for fault tolerant applications.

12.3. Cronus

Cronus is an operating system designed and implemented at BBN Laboratories. Some of the salient points of Cronus is 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 [BeRe85, GuDe86, ScTh86].

Like Eden, Cronus uses the active objects. This is necessary to be able to make Cronus run on top of Unix, and be an added function to Unix programs. 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.

12.4. ISIS

ISIS 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 [Bi85A, Bi85B].

12.5. 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 Sun-3 computers, networked over Ethernets. 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].

The key design criteria for ArchOS and Alpha are time critical computations and not reliability. Fault tolerance is not an issue, as the operating conditions are more susceptible to total failure rather than partial failure.

12.6. 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 confirm to the client server paradigm. A client sends a request to the server, and the client blocks untill the server replies [ChZw83].

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.

12.7. Mach

Mach has been developed at Carnegie Mellon, and looks like a Unix extension. Though Mach is not implemented "on top of Unix" it is implemented to look like distributed Unix. Mach is compatible with Unix at the object code level, that is Mach supports all system calls supported by Unix, and hence compiled Unix code can run on Mach. Mach uses the Accent message operating system as its base layer, and Accent provides the communication support. In addition Mach provides support for multiprocessors and distributed systems, memory mapped files, processing abstractions called *tasks* and *threads* [Ac86].

The activity in Mach is carried by tasks and threads. A task is similar to a Unix process. It is an address space and an execution environment. A task may be composed of several threads. A thread is a thread of control that can concurrently execute with other threads as a part of the same task, in the tasks address space. Messages are typed data that can be used by threads to communicate, and messages are routed through ports. Ports are addressable through capabilities.

The approaches used by Mach and *Clouds* are conceptually different and it is hard to draw conclusions about the differences in capabilities and usabilities at this stage. Mach however does not provide transaction support.

13. Concluding Remarks

Clouds provides an ideal 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 a 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, without the performance penalty of running *Clouds* 'on top of' Unix (or conversely). 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 experimenting 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 maybe redesign the system to fit the needs for distributed computing, and thus give rise to wider usage of distributed systems.

14. 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. Khaldi, *A Multicast Scheme for Locating Objects in a Distributed System*. Techical Report GIT-ICS-87/01, School of Information and Computer Science, Georgia Tech, January 1987.

[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.

[AlMc82] J. E. Allchin 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, Atlanta, GA, 1982.

[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.

[Bi85A] 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).

[Bi85B] K. P. Birman, *Replication and Fault-Tolerance in the ISIS System*, ACM SIGOPS, Proceedings of the Tenth Symposium on Operating Systems Principles, December 1985 Orcas Island, Washington, (Also released as technical report TR 85-668).

[ChZw83] D. R. Cheriton and W. Zwaenepoel, *The Distributed V Kernel and its Performance for Diskless Workstations*, Proceedings of the Ninth Symposium on Operating Systems Principles, ACM SIGOPS, Bretton Woods, NH, October 1983.

[Da86] P. Dasgupta, *A Probe-Based Fault Tolerant Scheme for the Clouds Operating System*, Technical Report GIT-ICS-86/05 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986.

[DaLe85] 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.

[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.

[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 Editing System*, (Extended Abstract) Conference Record of the Thirteenth Symposium on Principles of Programming Languages, ACM SIGACT/SIGPLAN, January 1986, St. Petersburg Beach, FL.

[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.

[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).

[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.

[Mc84A] 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).

[Mc84B] M. S. McKendry, *Fault-Tolerant Scheduling Mechanisms*, (Unpublished Technical Report), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, May 1984, (Draft only).

[Mc85] M. S. McKendry, *Ordering Actions for Visibility*, Transactions on Software Engineering, IEEE (11,6) June 1985 (Also released as technical report GIT-ICS-84/05).

[McAl83] M. S. McKendry, J. E. Allchin and W. C. Thibault, *Architecture for a Global Operating System*, IEEE Infocom, April 1983.

[Mo81] J. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[MuMo83] E. T. Mueller, J. D. Moore and G. J. Popek, *A Nested Transaction Mechanism for LOCUS*, Proceedings of the Ninth Symposium on Operating Systems Principles, ACM SIGOPS, Bretton Woods, NH, October 1983.

[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.

[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).

[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).

[SpBu84] A. Z. Spector, J. Butcher, D. S. Daniels and others, *Support for Distributed Transactions in the TABS Prototype*, July 1984, Technical Report CMU-CS-84-132, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

[WaPo83] B. Walker, G. Popek, R. English, C. Kline and G. Thiel, *The LOCUS Distributed Operating System*, Proceedings of the Ninth Symposium on Operating Systems Principles, Bretton Woods, NH, ACM SIGOPS, pp. 49-70, October 1983. (Available as *Operating Systems Review* 17, no. 5)

[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*, 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).

[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.