The Design and Implementation of the *Clouds* Distributed Operating System *

P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu–Aubán, P. W. Hutto, M. Y. A. Khalidi[†], *C. J. Wilkenloh*

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract

Clouds is a native operating system for a distributed environment. The *Clouds* operating system is built on top of a kernel called *Ra*. *Ra* is a second generation kernel derived from our experience with the first version of the *Clouds* operating system. *Ra* is a minimal, flexible kernel that provides a framework for implementing a variety of distributed operating systems.

This paper presents the *Clouds* paradigm and a brief overview of its first implementation. We then present the details of the *Ra* kernel, the rationale for its design, and the system services that constitute the *Clouds* operating system.

Keywords: Distributed Operating Systems, Distributed Computing, Operating Systems.

1 Introduction

Clouds is a distributed operating system built on top of a minimal kernel called *Ra*. The paradigm supported by *Clouds* provides an abstraction of storage called *objects* and an abstraction of execution called *threads*.

1.1 Basic Philosophies

A primary research topic of the *Clouds* project is the development of a set of techniques that can be used to construct a simple, usable distributed operating system. A distributed operating system should integrate a set of loosely-coupled machines into a centralized work environment. The following are the requirements of such a distributed system:

- The operating system must integrate a number of computers, both compute servers and data servers, into one operating environment.
- The system structuring paradigm is important in deciding the appeal of the system. This should be clean, elegant, simple to use and feasible.
- A simple, efficient, yet effective implementation.

To attain this end, we proposed the following basic philosophies:

*This work funded by NSF grant CCR-8619886
[†]Now at Sun Microsystems

- We use the much advocated *minimalist philosophy* towards operating system design. The operating system is divided into several clean, well defined modules and layers. The kernel is one such layer of the operating system and supports only features that cannot be supported elsewhere. Modularity in these layers limits the amount of interference (side-effects) and gives rise to easier upgrading and debugging.

- Three basic mechanisms common to all general-purpose systems are computation, storage and I/O. We use simple primitives to support these: namely, light-weight processes and persistent object memory. Light-weight processes are involved in computation. Persistent object memory serves for all storage needs. The need for user-level disk I/O in our model is eliminated. Terminal I/O is provided as a special case of object access.

1.2 *Clouds* Design Objectives

Clouds is designed to run on a set of general purpose computers (uniprocessors or multiprocessors) that are connected via a local-area network. The major design objectives for *Clouds* are:

- Integration of resources through cooperation and location transparency, leading to simple and uniform interfaces for distributed processing.

- Support for various forms of atomicity and data consistency, including transaction processing, and the ability to tolerate failures.

- Portability, extensibility and efficient implementation.

Clouds coalesces a distributed network of computers into an integrated computing environment with the look and feel of a centralized timesharing system. In addition to the integration, the paradigm used for defining and implementing the system structure of the *Clouds* system is a persistent object/thread model. This model provides threads to support computation and objects to support an abstraction of storage. The model has been augmented to to provide support for reliable programs [CD89, Che90]. The consistency techniques have been designed, but not implemented, and are outside the scope of this paper.

The rest of this paper is organized as follows. An overview of the *Clouds* project is provided in section 2, followed by an overview of the *Clouds* paradigm in section 3. The paradigm is the common link between the first implementation of *Clouds* (*Clouds v.1*) and the current version (*Clouds v.2*). We present the implementation of *Clouds v.1* in section 4, the structure and implementation of *Clouds v.2* in more detail in section 5, and the reasons behind the *Clouds* redesign in section 6. Section 7 presents some of details of the *Ra* implementation and section 8 describes the current state of the implementation of *Clouds* on *Ra*.

2 Project Overview

The first version of the *Clouds* kernel was implemented in 1986. This version is referred to as *Clouds v.1* and was used as an experimental testbed by the implementors. This implementation was successful in demonstrating the feasibility of a native operating system supporting the object model. Our experience with *Clouds v.1* provided insights into developing better implementations of the object/thread paradigm.

The lessons learned from the first implementation have been used to redesign the kernel and build a new version of the operating system called *Clouds v.2*. *Clouds v.2* uses an object/thread paradigm which is derived from the object/process/action system [All83, Spa86, Wil87] used by *Clouds v.1*. However, most of the design and implementation of the system are substantially different. *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.

The present implementation of *Clouds* supports the persistent object/thread paradigm, a networking protocol, distributed virtual memory support, user-level I/O with virtual terminals on Unix,[1] a C++ based programming language and some system management software.

Work in progress includes user interfaces, consistency support, fault tolerance, language environments, object programming conventions and better programming language support for object typing, instantiation, and inheritance.

3 The *Clouds* Paradigm

All data, programs, devices and resources in *Clouds* are encapsulated in objects. Objects represent the passive entities in the system. Activity is provided by threads, which execute within objects.

3.1 Objects

A *Clouds* object, at the conceptual level, is a virtual address space. Unlike virtual address spaces in conventional operating systems, a *Clouds* object is persistent and is not tied to any thread. A *Clouds* object exists forever and survives system crashes and shutdowns (like a file) unless explicitly deleted. As will be seen in the following description of objects, *Clouds* objects are somewhat "heavyweight" and are better suited for storage and execution of large-grained data and programs because invocation and storage of objects bear some non-trivial overhead.

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 was Hydra [WCC+74, WLH81].) The contents of each virtual address space are protected from outside access so that memory (data) in an object is accessible only by the code in that object and the operating system.

Each object is an encapsulated address space with entry points at which threads may commence execution. The code that is accessible through an entry point is known as an object *operation*. Data cannot be transmitted in or out of the object freely, but can be passed as parameters (see the discussion on threads in section 3.2).

Each *Clouds* object has a global system-level name called a *sysname*, which is a bit-string that is unique over the entire distributed system. Sysnames do *not* include the current location of the object (objects may migrate). Therefore, the sysname-based naming scheme in *Clouds* creates a uniform, flat system name space for objects, and allows the object mobility needed for load balancing and reconfiguration. User-level names are translated to sysnames using a nameserver.

Clouds objects are programmed by the user. System utilities can be implemented using *Clouds* objects. A complete *Clouds* object can contain user-defined code and data, and system-defined code and data that handles synchronization and recovery. In addition, it contains a volatile heap for temporary memory allocation, and a permanent heap for allocating memory that becomes a part of the data structures in the object. Locks and sysnames of other objects are also a part of the object data space.

3.2 Threads

The only form of user activity in the *Clouds* system is the user thread. A thread can be viewed as a thread of control that runs code in objects, traversing objects and machines as it executes. A thread executes in an object by entering it through one of several entry points; after the execution is complete the thread leaves the object. The code in the object can contain a call to an operation in another object with arguments. When the 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 with returned results. These arguments/results are strictly data; they may not be addresses. (Note that sysnames are data.)

[1] Unix is a trademark of AT&T.

This restriction is necessary as addresses which are meaningful in the context of one object are meaningless in the context of another object. In addition, object invocations can be nested.

Several threads can simultaneously enter an object and execute concurrently (or in parallel, if the host machine is a multiprocessor). Multiple threads executing in the same object share the contents of the object's address space.

Unlike processes in conventional operating systems, a thread can execute code in multiple address spaces. Visibility within an address space is limited to that address space. Therefore, a thread cannot access any data outside its current address space. Control transfer between address spaces occurs through object invocation, and data transfer between address spaces occurs through parameters.

3.3 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 long-lived modules which do not share global data. Machine boundaries are transparent to inter-object procedure calls or invocations. Local invocations and remote invocations are differentiated only by the operating system.

The storage mechanism used in *Clouds* differs from those found in conventional operating systems. Conventionally, files are used to store persistent data. Memory is associated with processes and is volatile. The contents of memory associated with a process are lost when the process terminates. Objects in *Clouds* unify the concepts of persistent storage and memory to create the concept of a persistent address space. This unification makes programming paradigms simpler.

Although files can be implemented using objects (a file is an object with operations such as read, write, seek, and so on), the need for having files disappears in most situations. Programs do not need to store data in file-like entities, since they can keep the data in the data spaces of objects, structured appropriately. The need for user-level naming of files transforms to the need for user-level naming of objects. Also, *Clouds* does not provide user-level support for disk I/O, since files are not supported by the operating system. The system creates the illusion of a large memory space that is persistent (non-volatile). Since memory is persistent, the need for files to store persistent data is eliminated.

In the object/thread paradigm, the need for messages is eliminated. Like files, 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. An arbitrary number of threads may execute concurrently within an object. Thus, memory in objects is inherently sharable memory.

The system therefore looks like a set of *persistent* address spaces, comprising what we call *object memory*, which allow control to flow through them. Activity is provided by threads moving among the population of objects through invocation (figures 1 and 2). The flow of data between objects is supported by parameter passing.

3.4 Objects and Types

At the operating system level there is only one type of object: *Clouds object*. A *Clouds* object is an address space which may contain one (or more) data segment(s) and one (or more) code segment(s). At the operating system level, the *Clouds* object has exactly one entry-point. *User objects* are built using *Clouds* objects via an appropriate language and compiler. User objects have multiple user-defined entry points and are implemented using the single entry point, operation numbers and a jump table.

User objects and their entry points are typed. Static type checking is performed on the object and entry point types. No runtime type checking is done by *Clouds*. Thus, type checking is implemented completely at the language level and enforced by compilers. A user object may contain several language defined object classes and object instances. These are completely contained within the user object and are not visible to the operating system. Using this object/thread paradigm, programmers can define a set of objects that encapsulate the application at hand.

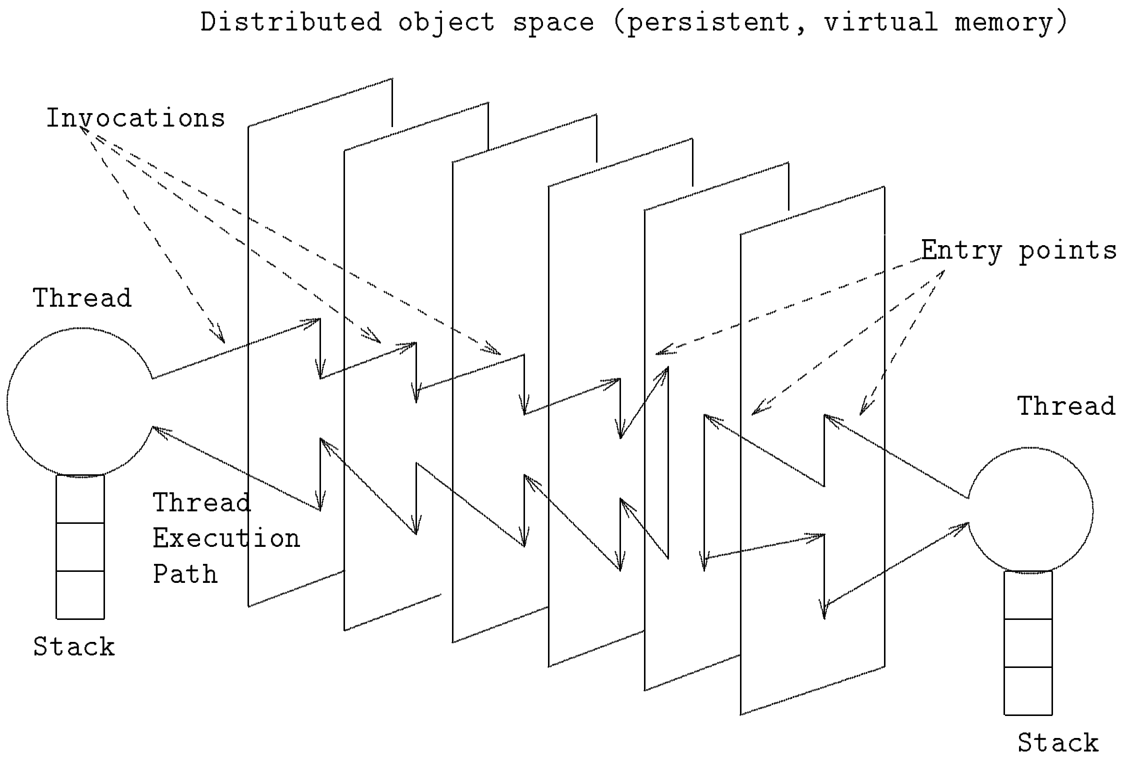


Figure 1: Object Memory in *Clouds*

Currently, we define user objects in an extended C++ language. The language supports single inheritance on *Clouds* user objects using the C++ object structuring paradigm.

4 *Clouds v.1*

The first implementation of a kernel for *Clouds* was finished during 1986 and is described in [Spa86, Pit86]. The kernel was broken up into four subsystems: object management, storage management, communications, and action management. A kernel-supported extension of the nested action model of Moss [Mos81, All83, Ken86] made it possible for the programmer to customize synchronization and recovery mechanisms with a set of locking and commit tools.

Since one of the goals of *Clouds* was to produce an efficient and usable system, a direct implementation on a bare machine (VAX-11) was preferred to an implementation on top of an existing operating system such as UNIX. Also most of the functionality needed by *Clouds* did not exist on Unix (such as persistent shared memory, threads) and most of the features provided by Unix were not needed for *Clouds*. The main goal of the implementation effort was to provide a proof of feasibility of implementing the object/thread model. The kernel, notwithstanding some of its drawbacks, was a successful demonstration of the feasibility of the *Clouds* approach, both in terms of implementation and use.

4.1 Objects

The basic primitives provided by the *Clouds v.1* kernel are processes and passive objects. Indeed, one kernel design objective was to support passive objects at the lowest possible level in the kernel [Spa86]. The main mechanism provided by the kernel was object invocation. The system relied heavily on the VAX virtual memory system to provide object support.

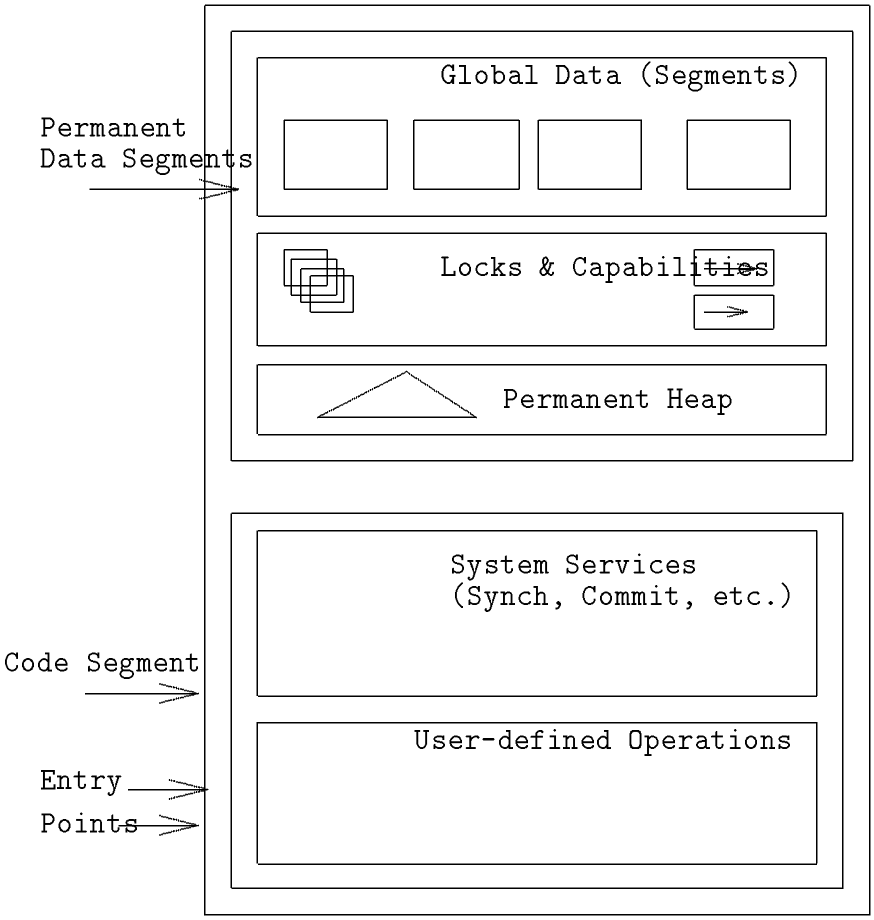


Figure 2: Structure of a *Clouds* Object

Passive objects in *Clouds v.1* are implemented as follows: the VAX virtual address space is divided into three segments by the system architecture, namely the P0, P1 and System segments. The System segment is used to map the kernel code. The P1 segment is used to map the process stack and the P0 segment is used to map the object space. The object space resides on secondary storage and page tables are used to map the object under invocation into the P0 segment directly from disk. Each process has its own pageable process stack.

The above scheme allows processes to traverse several object spaces, while executing, by simply remapping the P0 segment of a process to contain the appropriate object space upon each invocation (see the next section for further details). Also, several processes are allowed to concurrently execute within the same object.

4.2 Object Invocation

The basic mechanism used in the *Clouds* kernel to process an object invocation from a thread t executing in object O_1 is the following: t constructs two argument lists, one for transferring arguments to the object being invoked, O_2, and the other to receive the output parameters (results) from the invocation of object O_2 (see [Spa86] for more details). After the construction of the argument lists, thread t enters the kernel through a protected system call or trap. The kernel searches for the object locally and, if found, uses the information in the object descriptor to construct the page mappings for the P0 segment. Then, the kernel saves the state of the thread, copies the arguments into the process space (P1 segment) and sets up the new mappings for the P0 segment. At this point, the contents of O_2 are accessible through the mappings of the P0 segment, and t can proceed with the invocation of O_2's method.

On return from the invocation, the thread t also builds an argument list with the return parameters, and then enters the kernel by means of a protected system call. The kernel now saves the parameter in a temporary area, sets up the P0 segment mappings for O_1, restores the saved state of t, and copies the return parameters wherever specified by the second argument list constructed by the thread at invocation time.

If upon invocation, the kernel cannot find object O_2 locally, it tries to find it remotely. To do so, it broadcasts an RPC request. The RPC server in the node that has O_2 acknowledges the invocation request, and creates a local slave process to invoke O_2 on behalf of t. The slave process then proceeds to invoke O_2 locally, and when the invocation completes, it sends the return arguments back to the invoking node. Then the kernel goes on to process the return parameters as described above.

4.3 Experiences with *Clouds v.1*

Our experiences with *Clouds v.1* were gained over a period of 6-8 months while attempting to build a user environment on top of the kernel. During this time, we encountered a number of problems in the kernel that had to be addressed. Unfortunately, while doing so, we discovered that complex module interdependencies within the kernel made it extremely difficult to add necessary functionality or fix non-trivial bugs without introducing new bugs in the process. One reason for this was due to poor code structure. Also, direct concurrent access and update of data structures by routines in different subsystems within the kernel resulted in synchronization and re-entrancy problems.

The final problem involved a change of platform. For economic reasons, we wanted to move to Sun workstations. However we found that VAX dependencies that appeared throughout the kernel (such as programming the VAX bus) made the kernel virtually impossible to port to a new architecture without a complete re-write.

In hindsight, these problems are not entirely surprising. While portable kernels are desirable, portability was not emphasized during the design and implementation of *Clouds v.1*. We were more interested in producing a working prototype. Also, *Clouds v.1* was implemented in C. Given the lack of support in C for software engineering techniques, it should not have come as a surprise that our first attempt at structuring the internals of a kernel supporting persistent objects resulted in a less than optimal design.

The basic design and philosophy behind *Clouds v.2* is a direct result of the problems we encountered in working with the first kernel. The second kernel is a *minimal* kernel, designed with flexibility, maintenance, and portability in mind. A complete discussion of the similarities and differences between *Clouds v.1* and *Ra* is contained in section 6.

5 *Clouds v.2*

The structure of *Clouds v.2* is different from *Clouds v.1*. The operating system consists of a minimal kernel called *Ra*, and a set of system-level objects providing the operating system services. *Ra* [BHK+88, BHK+89] supports a set of basic system functions: virtual memory management, lightweight processes, low-level scheduling, and support for extensibility.

5.1 The Minimal Kernel Approach

A minimal kernel provides a small set of operations and abstractions that can be effectively used to implement portable operating systems *independent* of the underlying hardware. The kernel creates a virtual machine that can be used to build operating systems. The minimal kernel idea is similar to the RISC approach used by computer architects and has been effectively used to build message-based operating systems such as V [CZ83], Accent [RR81], and Amoeba [TM81]. The rule we attempted to follow in our design was: *Any service that can be provided outside the kernel without adversely effecting performance should not be included in the kernel.*

As a result, *Ra* is primarily a sophisticated memory manager and a low-level scheduler. *Ra* creates a view of memory in terms of segments, windows and virtual spaces. Unlike the *Clouds v.1* kernel, *Ra* does *not* support objects, invocations, storage management, thread management, device management, network protocols or any user services. All of these services are built on top of the *Ra* kernel as modules called *system objects*. System objects provide other systems services (user object management, synchronization, naming, networking, device handling, atomicity and so on) and create the operating system environment. Currently the *Ra* kernel and all of the essential system objects are operational; the project is now focusing on higher level services.

There are several advantages to the use of a minimal kernel. The kernel is small, hence easier to build, debug, and maintain. Minimal kernels assist in the separation of mechanisms from policy which is critical in achieving operating systems flexibility and modularity [WCC+74]. A minimal kernel provides the mechanisms and the services above the kernel implement policy. The services can often be added, removed or replaced without the need for recompiling the kernel or rebooting the system.

5.2 The *Ra* Kernel

The principal objectives in *Ra*'s design were:

- *Ra* should be a small kernel that creates a logical view of the underlying machine.
- *Ra* should be easily extensible.
- One of the possible extensions of *Ra* should be *Clouds*.
- It should be possible to effect an efficient implementation on a variety of architectures.

In addition to the above, the implementation of *Ra* should clearly identify and separate the parts that depend on the architecture for which the implementation is being targeted. This should reduce the effort required to port the kernel to different architectures.

As was previously stated, object invocation in the first version of *Clouds* was implemented by manipulating the virtual memory mappings. The *Clouds v.1* kernel was targeted to support object invocations. *Ra* is targeted to support the memory management needs of objects. From our

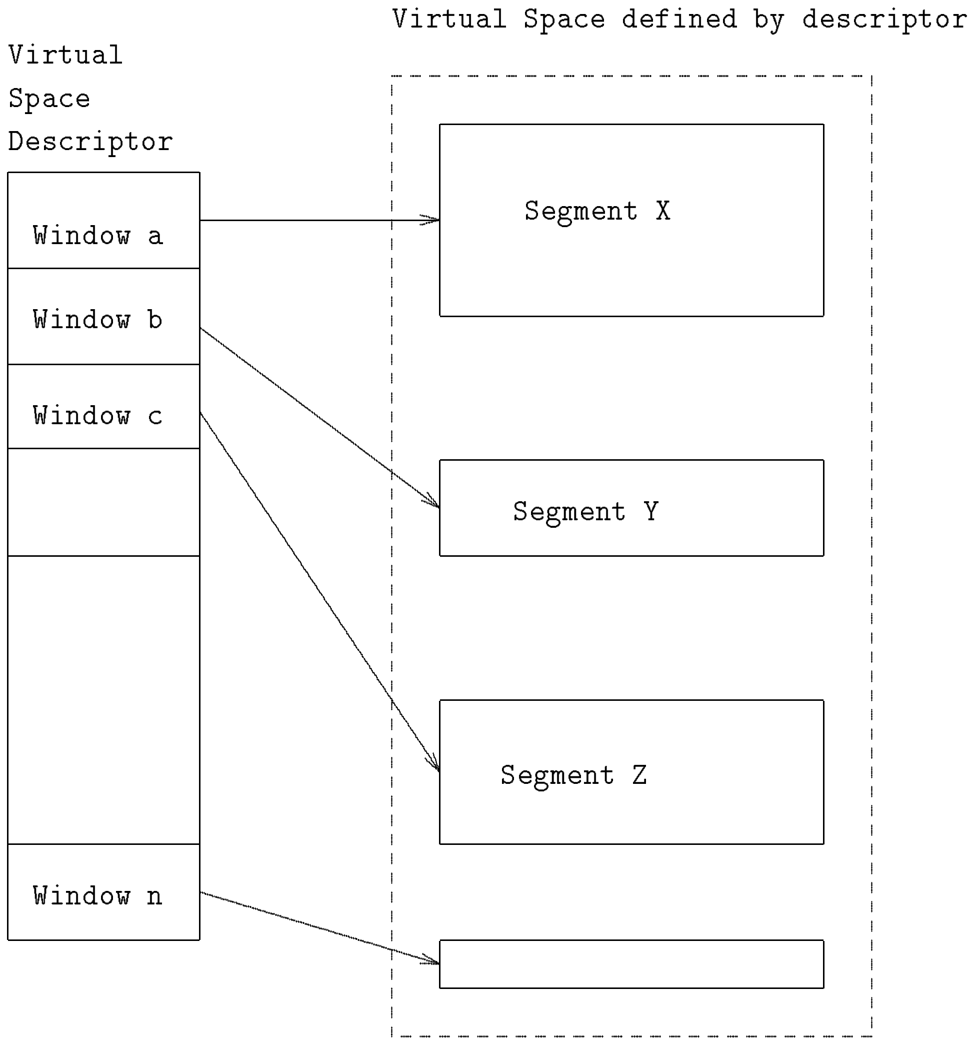


Figure 3: VSD Defining a Virtual Space

experience with the first *Clouds* kernel, we identified generalizations in the virtual memory management mechanisms that we felt would provide a larger degree of flexibility in the the design and implementation of new systems.

Memory is the primary abstraction in the *Clouds* paradigm. Since objects are implemented as autonomous address spaces, they need to be built out of sharable, pageable segments. These segments should be easily migratable to support distributed memory. These requirements led to the design of the virtual memory architecture of *Ra*.

Ra supports a set of primitive abstractions and an extension facility. The three major abstractions are:

IsiBas

An *IsiBa* [2] is an abstraction of activity, and is basically a very lightweight process. The IsiBa is simply a schedulable entity consisting of a PCB. An IsiBa has to be provided with a stack segment and a code segment before it is scheduled. Therefore, IsiBas can be used to create

[2]The term IsiBa comes from early Egyptian. Isi = light, Ba = soul and was coined by the early designers of the *Ra* kernel. It is now felt that the term is confusing and we periodically agree to change it but have not been able to agree on a replacement term that is both informative and non-boring. The choice of a replacement term is a recurring topic of animated discussion.

user-level processes, threads, kernel processes, daemons and can be used for a host of tasks needing activity.

Segments

Segments conceptualize persistent memory. A segment is a contiguous block of uninterpreted memory. Segments are explicitly created and persist until destroyed. Each segment has a unique system-wide sysname, and a collection of storage attributes. Segments are stored in a facility called the partition. *Ra* does not support partitions, but assumes the existence of at least one. Partitions are described later. *Ra* provides a set of functions to manipulate segments (create, extend, install, page-in/out and so on).

Virtual Spaces

A virtual space abstracts a fixed-size contiguous region of an IsiBa's virtual address space. Ranges of memory in a virtual space can be associated with (or mapped to) an arbitrary range of memory in a segment. Each such mapping (called a *window*) also defines the protections (user-level read, read-write, kernel-level read, etc.) on the defined range of memory. Virtual spaces are defined and controlled by a *Virtual Space Descriptor* or *VSD* for short (see figure 3). Virtual spaces can be associated with an IsiBa. This is called *installing* a virtual space. *Ra* is responsible for ensuring that the state of the IsiBa's hardware address space corresponds to that defined by its installed virtual spaces (the virtual memory architecture is described in more detail in section 5.3). Virtual spaces can be used by higher level routines to implement such things as objects. *Ra* supplies functions that assemble, install and manipulate virtual spaces.

Ra is implemented using the C++ language and heavily uses the object oriented programming paradigm provided by C++. The functional units in *Ra* are encapsulated in C++ objects and C++ classes are used to provide structure to the implementation. C++ classes are used, for example, to define base classes (and hence minimal interfaces) for system objects, devices, partitions, virtual memory managers, etc.

Ra is related to the Mach kernel [ABB+86] by its some of its virtual memory mechanisms and its approach to portable design [RTY+87]. The Choices kernel [CJR87] is built using object-oriented programming techniques and is thus related to the implementation structure of *Ra*. During the design of *Ra*, we were also influenced by ideas from Multics [Org72], Hydra [WCC+74], and Accent [RR81], as have other distributed operating systems designed in the tradition of *Clouds* [NKK86, Nor87].

5.3 Virtual Memory Architecture

Ra provides a two-level virtual memory architecture similar to *Clouds v.1* (similarities and differences are discussed in section 6). *Ra* breaks down the machine's virtual address space, hereafter referred to as the *hardware address space*, into a number of fixed-sized, contiguous, non-overlapping regions. These regions are used to map different types of virtual spaces.

5.3.1 Virtual Spaces

Ra supports three types of virtual spaces: O, P, and K (see figure 4). The contents of each virtual space (of any type, O, P, or K) is described by a set of *windows* in the virtual space descriptor. Each window w associates protection with a range $(x, x+n)$ of virtual memory controlled by the virtual space and associates a range $(y, y+n)$ of bytes in a segment s into the virtual memory range $(x, x+n)$. An access of byte $x+a$ in window w will reference byte $y+a$ in segment s. A virtual space V may contain an arbitrary number of windows mapping non-overlapping ranges within the bounds of the region controlled by the virtual space.

Instances of virtual spaces of type O (O space) are intended to map shared entities (e.g. objects), and spaces of type P (P space) are intended to map process private entities (e.g. stacks). In a single-processor system there can be only one instance of a virtual space of type K (K space). This is used to map the *Ra* kernel and the system objects. There are additional operations that K space descriptors

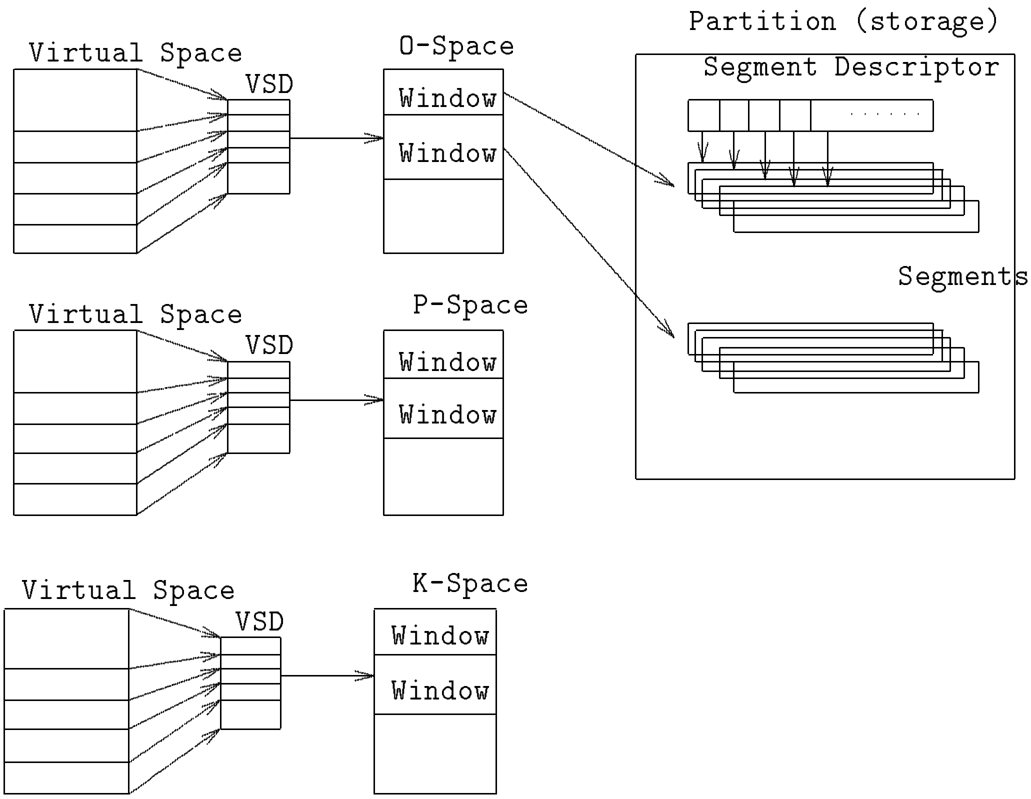


Figure 4: Hardware Address Space

must support in addition to the standard operations defined by the virtual space class. This is handled by making the Kernel Virtual Space Descriptors (KVSD) a derived class (subclass) from the base class VSD.

Each IsiBa can have one O space, one P space, and one K space currently installed. In the current uniprocessor implementation, every IsiBa has the same K space installed, effectively mapping the kernel into the address space of every IsiBa.

5.3.2 Shared Memory

Sharing memory occurs when two virtual addresses refer to the same piece of physical memory. In the *Ra* kernel, sharing can take place at two levels. Two IsiBas may share memory at the virtual space level by installing the same P or O (or both) virtual space. When they reference memory within the bounds of the shared virtual space, they will automatically reference the same piece of physical memory or backing store.

Memory may also be shared at window level. If virtual space *V1* has a window *w1* $(a, a+n)$ that maps to $(x, x+n)$ within the segment s and a virtual space *V2* has a mapped window *w2* $(b, b+m)$ that maps to $(y, y+m)$ within the same segment s and the two ranges overlap, then the overlapping area of memory will be shared by the two windows *w1* and *w2*. To take a simple case, if *w1* and *w2* both map to the same range of addresses in the segment s (that is, m = n), then a reference to address $a + z, (z < n)$ and a reference to address $b + z$ will both reference the same memory. Furthermore, it does not matter to the *Ra* kernel if *V1* and *V2* are the same virtual space (in which case memory aliasing will occur) or different virtual spaces, nor if *V1* and *V2* are different *types* of virtual spaces (which will also cause memory aliasing if both *V1* and *V2* are installed in the same IsiBa). Thus, two IsiBas may share memory by installing the same virtual space. Also they may share memory if they have different virtual spaces installed but the virtual spaces map common segments.

5.4 *Ra* Synchronization Primitives

The kernel defines a set of primitives to be used for synchronization and concurrency control inside the kernel and system objects. At the lowest level, *Ra* defines (machine-dependent) interrupt level control facilities. Since *Ra* is intended to be portable to multi-processors, spin locks are provided. These low-level facilities are used to construct kernel-level semaphores, memoryless events, and read/write locks.

5.5 Extensibility

The design of *Ra* abstracts a logical machine that provides virtual memory mechanisms and low level scheduling. Hence, it is unusable by itself, and must be extended using system objects into an operating system such as *Clouds*. System objects reside in kernel space (K space), *not* user space (O and P space), have direct access to data and code in the kernel, and share the same protection and privileges. Hence, system objects are not the same as *Clouds* objects. Some system objects are called *essential* system objects because they are essential to run *Ra*. For example, the *Partition* is an essential system object. *Ra* assumes the existence of at least one partition.

The system objects in *Clouds* are organized in a hierarchy of classes, ultimately deriving from the *SysObj* class. *Ra* defines a system object interface and each system object must adhere to this interface. The system object must have *initialize* and *shutdown* methods, can have private memory and can access kernel data structures though kernel classes.

Kernel classes are collections of kernel data and procedures to access and manipulate that data. These include the *VSD* class, *sysname* class, and *cpu* class. As viewed from a system object, the kernel is a collection of kernel classes and instances of these classes. Each cpu, for example, is a specific instance of the *cpu* class and can be manipulated using methods defined by the *cpu* class.

The system object interface enforces the strict adherence to modularity in the operating system. *Clouds* has been built by attaching all the relevant system objects to *Ra*. The system objects can be thought of as plug in software modules that can be linked in with the kernel or loaded dynamically into a running system.

5.6 Partitions

Partitions are repositories for segments. *Ra* requests segments from a partition when needed and releases them to the partition when their usage is over. The partition is responsible for managing segments on secondary storage. A partition must support at least the following operations: *ActivateSegment, DeactivateSegment, ReadPage*, and *WritePage*. A segment must be *activated* before being used, by calling the *ActivateSegment* operation. This allows the partition to set up any necessary in-memory data structures. *DeactivateSegment* is then called after the kernel has finished using the segment. *ReadPage* and *WritePage* are self-explanatory.

A partition is responsible for maintaining and manipulating segments. Each segment is maintained by exactly one partition, and the segment is said to *reside* in that partition. The partition in which the segment resides is called the *controlling* partition. Creation and deletion of segments is performed through their controlling partitions. The partition is responsible for maintaining the information which describes the segment in secondary storage, similar to file-system code in a conventional operating system. To read or write segment pages on secondary storage, the controlling partition of the segment is invoked. The partition is notified that one of its segments will be subject to further activity by activating the segment. Similarly, the partition is told that a segment will not be used in the near future by deactivating the segment.

In our implementation, a network disk partition stores segments on a Unix file-server. Each segment is stored as a Unix file. The partition provides pages of segments to *Ra* over a network. The local *Ra* kernel is unaware of this mechanism. The network disk partition provides an easy way to create segments on Unix which are then available to *Clouds* machines. This partition is also used for paging and swapping activity.

5.7 Distributed Shared Memory

Segments residing in a network disk partition cannot be shared by multiple machines. Thus each network disk partition creates a private storage space for each *Clouds* machine.

However, to allow distribution, all objects should be accessible by all *Clouds* machines. This means the segments should be stored by a set of data-servers and should be accessible by all compute-servers. This is allowed by Distributed Shared Memory Partitions (DSM Partitions).

Sharing segments among machines may lead to multiple cached copies of the segments. To ensure one-copy semantics used by the *Clouds* paradigm, a coherence controller is necessary. The DSM partition implements a set of protocols which enforces the coherence of shared segments (figure 5). Each DSM partition knows about a set of segment managers called DSM controllers. Currently, DSM controllers run on Unix machines and store segments as files. When *Clouds* accesses a segment (due to the invocation of an object that uses that segment), the DSM partition on the local machine gets the segment from the DSM controller. The DSM controller runs coherence algorithms which ensure that there is only one copy of the segment in the *Clouds* system [RAK89, Kha89]. This makes memory appear to be one-copy shared and creates the view that all objects reside on all machines.

6 *Ra* and *Clouds v.1*

The design of *Ra* draws heavily on our experiences with *Clouds v.1*. The *Clouds v.1* was essentially a monolithic kernel. Expandability and flexibility were not primary design goals.

As every academic/commercial operating systems evolves, new sets of researchers and implementors add to the design and functionality in ways not anticipated by the original designers. While attempting to build a user environment for *Clouds v.1*, we realized that it was difficult to extend the system. A number of features in *Ra* attempt to address this problem.

Ra supports a more primitive set of abstractions than *Clouds v.1*, however, the system object class allows the addition of operating system services not supported by the kernel. System objects

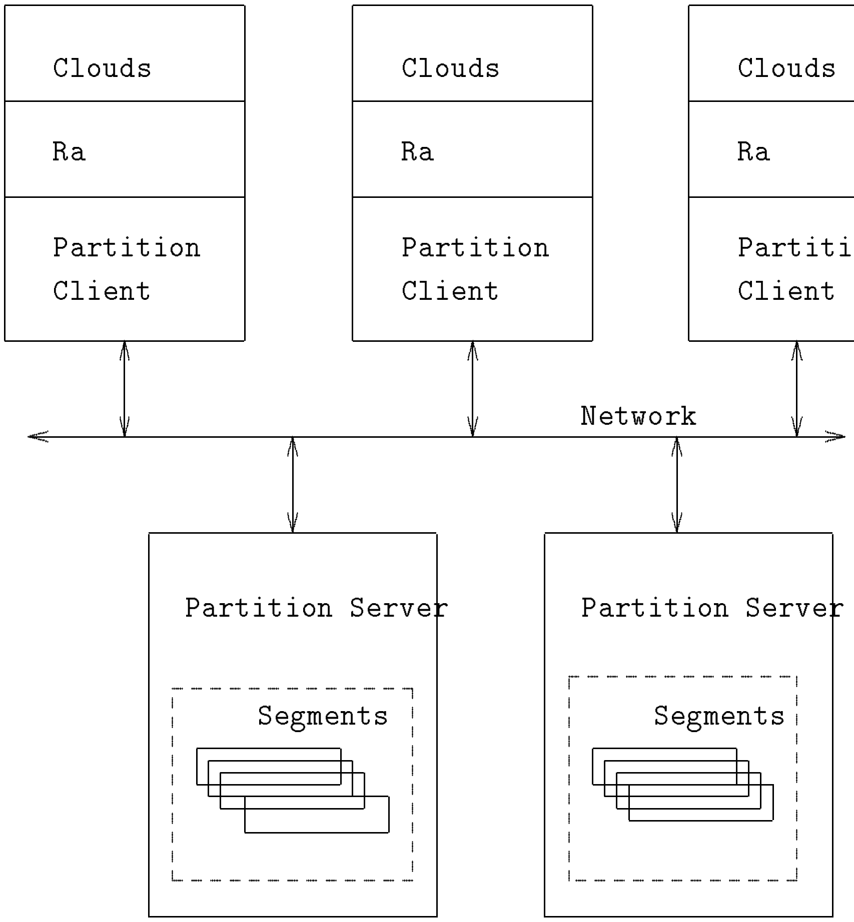


Figure 5: DSM Partitions

allow flexibility and enforce modularity. This structuring technique is is worthy of note in that while many systems allow extensions in the form of device drivers, far more fundamental operating system services have been implemented as *Ra* system objects.

To support testing of higher-level algorithms, the *Ra* design also strives to separate mechanisms that implement actions from policies that decide when actions should be taken. Furthermore, the object-oriented structure of the *Ra* kernel lends itself to easier maintenance than the *Clouds v.1* kernel.

6.1 Multi-threading

In the *Clouds v.1* design, processes were somewhat heavyweight and the kernel was basically passive. This meant that virtually all operating system services were provided through interrupt and fault handlers. Therefore, the entire kernel had to be multi-threaded. This necessitated very delicate handling of interrupt priority levels and synchronization primitives throughout the entire kernel. The problem was compounded by the fact that concurrency-related mistakes usually result in non-reproducible timing-related errors which are very difficult to debug. Unfortunately, in a 22,000 line multi-threaded kernel written from scratch by a handful of people, it is virtually guaranteed that concurrency-related errors will creep in.

In *Ra*, IsiBas can be used to implement kernel daemons. Kernel daemons run in kernel space with kernel privileges. These daemons can be used to implement single-threaded servers that are dispatched using interrupts. This reduces the amount of code that has to be multi-threaded and run with interrupts masked. Kernel daemons have been used in a number of places, most notably in the implementation of remote procedure calls, timer services, network protocols and DSM.

6.2 Virtual Memory Architecture

The design and implementation of *Ra*'s virtual memory architecture incorporates a generalization of the virtual memory architecture of *Clouds v.1* and portability-oriented structuring ideas found in Mach [RTY+87]. Rather than embed support deeply in the kernel for one notion of how objects could be structured, the decision was made to support a flexible, sophisticated virtual memory architecture that could be adapted to more than one object implementation. This would allow future developers the freedom to modify the object implementation without having to re-write kernel internals.

Ra thus supports the notion of an O space, P space, and K space. By virtue of using the VAX P0, P1, and System segments to contain objects, the process stack, and the kernel, respectively, *Clouds v.1* also supported the equivalent of an O, P, and K space. However, since support for objects, processes, and actions was designed into the kernel at the lowest possible level, the structuring of each type of space was hard-coded into the kernel. The P0 segment was broken up into windows but the P1 and System segments were not. In *Ra*, the structure of the *Clouds v.1* P0 segment is generalized into the notion of a virtual space and applied in an orthogonal fashion to all three types of virtual spaces. This generalization leads to a number of advantages.

By structuring the hardware address space as virtual spaces, it is possible in *Ra* to break the address space up into more than three virtual spaces. While the original designers foresaw only the need for an O, P, and K space, our recent work in memory semantics has convinced us of the need to add a fourth virtual space [DC90, Che90]. Due to the uniform handling of virtual spaces in the kernel, adding another virtual space will be straightforward.

The fact that P space can be broken up into windows allows more flexibility in designing and implementing user processes and object invocation. The original *Clouds v.1* design placed all non-stack per-process data in P0 space. The current user object implementation could have been implemented similarly. However, it was decided to have all per-process data (stack, heap, parameter passing area) reside in P space instead, which means the entire object space can be shared. This allows all threads executing in the same object to share the same O space virtual memory tables. The *Ra* kernel is capable of supporting either implementation.

The K space windows allow the kernel and system objects to use the window class to map portions of segments into well-defined address ranges, operate on the data there, flush the changes out and unmap the window (freeing it up for further use). This facility is used by the user process controller to set up a new processes and freeze/restore them (see section 8.2).

Also, the *Ra* virtual memory system allows for the existence of more than one *virtual memory manager*. Virtual memory managers handle page faults. Each *Ra* segment can have a segment type and each segment type can have its own virtual memory manager. When a page faults occurs, the kernel determines the segment being accessed by the fault. If the segment has its own virtual memory manager, the kernel calls that manager. Otherwise, the kernel calls a default virtual memory manager to service the fault. This design allows the complexity of services such as shadow-based recovery to be isolated in its own virtual memory manager.

Finally, unlike the *Clouds v.1* virtual memory system, the *Ra* virtual memory system is designed with machine independent and machine dependent classes (inspired by Mach). Each machine independent class that may have to rely extensively on machine-dependent details has a corresponding machine-dependent class that presents an abstraction of the underlying hardware to the rest of the kernel. This isolates the machine dependencies from the rest of the kernel.

7 Implementation of *Ra*

Ra is implemented in C++ [Str86] on the Sun-3/60 architecture. C++ was chosen over C due to the extra support for data abstraction, and object-oriented design.

C++ facilities such as private data and methods, derived classes, and virtual functions make it easier to write modular, layered code and hide the implementation details of one part of the kernel

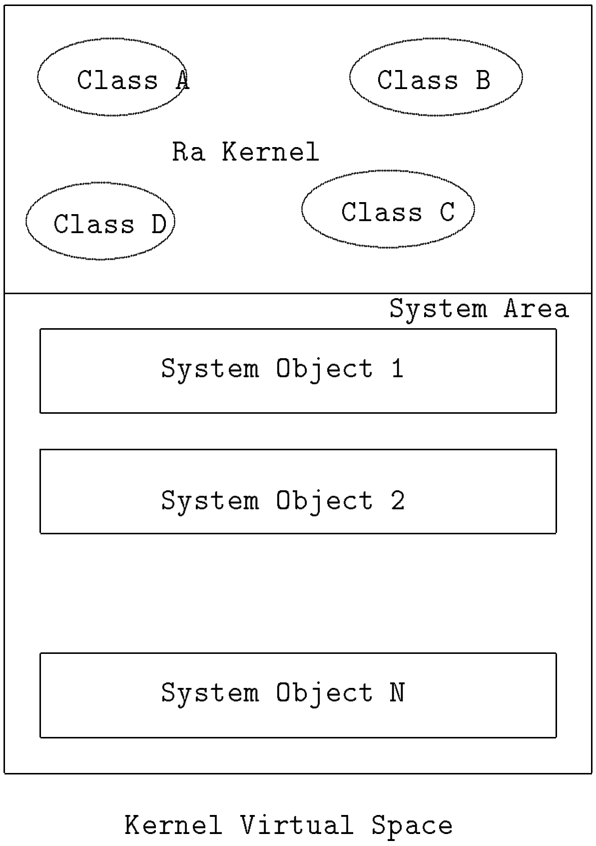


Figure 6: System Objects and System Space

from the rest. This, in turn, reduces hidden interdependencies which makes it easer to change parts of the system without introducing unforeseen side-effects. While all good software designers strive to do this, the language support (and enforcement) provided by C++ has made this a much easier task. Furthermore, the C++ inline function facility makes it possible to write highly-layered/modular code without incurring extra function-call overhead.

Thus, while the kernel internals are quite intricate, well-defined interfaces exist for requesting kernel services. C++ type-checking enforces adherence to the defined interfaces and hence prevents system implementors from by-passing those interfaces, but the ability to define optional parameters in class methods enables interfaces to be easily extended while retaining compatibility with existing code.

The design and implementation of *Ra* is designed to identify and isolate machine dependencies. Like the virtual memory system, *Ra* is is divided into two sets of files containing machine dependent and machine independent classes and definitions.

The implementation of the *Ra* kernel consists of about 1,000 lines of assembly code and 12,000 lines of C++ code. Approximately 6,000 lines are machine dependent code while the rest are machine independent. In addition, 17,000 lines of C++ code have been added to the system in the form of system objects.

8 *Clouds v.2* and *Ra*

Clouds v.2 consists of the *Ra* kernel plus a collection of system objects implementing *Clouds* semantics (see figures 6 and 7).

The system objects contain more code than the *Ra* kernel itself. A complete description of

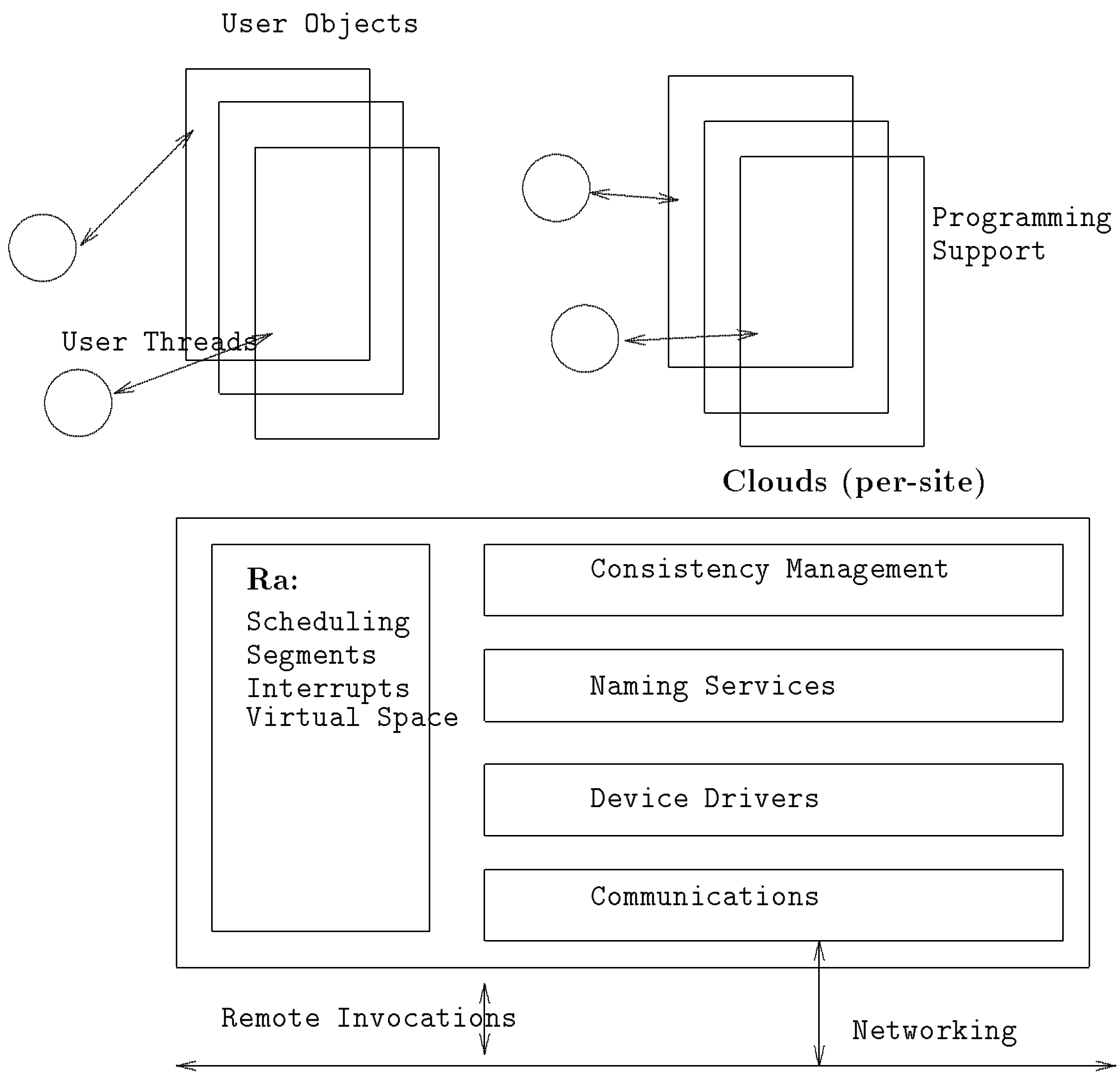


Figure 7: The *Clouds/Ra* Environment

these are beyond the scope of the paper. The system objects currently in operation include: buffer manager, user I/O manager, tty manager, Ethernet driver, *Ra* Transport Protocol, network disk partition, DSM partition, object manager, thread manager, process controller, RPC controller and system monitor. Many of the system objects are implemented using kernel daemons. Some of the key functions of the system objects are described below.

8.1 Object Management

An object is implemented in *Ra* by using a virtual space (O space). The storage of the object is ultimately accessed via the segments mapped by the windows of the virtual space. The *sysname* of the object is the *sysname* of a segment containing the windowing information (VSD). Objects are always mapped into the O space when being executed. A *Clouds* object is a special case of a *Ra* virtual space.

The object invocation mechanism is implemented by means of two system objects: the *Process Controller* and *Object Manager*. Processes are implemented using segments (to back the per-process stack for example) and are managed by the process controller. Objects are invoked by threads (implemented using processes). When a thread invokes an object, the process controller is responsible for saving/protecting the state of the current invocation and installing any state required by the new invocation. During this process, the process controller notifies the object manager that an object is about to be invoked (referenced) and that object manager is responsible for ensuring that the control information (virtual space descriptor) for the object exists. The object is installed in the O space of the thread and the thread is placed in the entry point. The physical location of the object is of no consequence as the DSM partition will page in the segments from the appropriate DSM controller (or server). Remote object invocation is performed through an RPC handler which communicates to an RPC server on a remote machine and asks it to invoke the requested object in a manner similar to the *Clouds v.1* object invocation mechanism.

Invoking the object locally and relying on DSM to page in the object across the network is not always more efficient than performing a remote procedure call. If processes on different nodes have the same locality of reference in an object and they invoke the same object using DSM to page the object, DSM will thrash, paging the common pages back and forth between the different nodes. In that case, it might be better to move all the computation to the same node by performing an RPC where the processes can physically share the memory.

In addition, if the load on the nodes of the system is not balanced, it may be a better idea to send the computation to a remote node. Notice that the object being invoked need not reside on the chosen node as DSM can be used to access the object's segments. This provides another mechanism besides process migration that can be used to balance the load on the system. However, using local invocations and DSM is superior to RPC in cases where there is a high degree of locality.

The policy that decides whether to execute a local invocation or a remote invocation has not been implemented. Currently the user can decide on the invocation mechanism by using appropriate system calls.

8.2 Thread Management

Processes are implemented by associating an IsiBa with a virtual space (installed in the P space) which controls per-process memory such as the process stack and the parameter passing areas. A thread is a process if the thread only uses local invocations, and is a collection of processes if the thread has performed a remote invocation. The thread manager keeps track of a thread's RPC calls (if any) and controls creation and termination of threads. However, thread managers do not directly manipulate processes or per-process memory. The process controller defines an abstraction of a process. The thread manager makes requests on the process controller to manipulate processes and to perform object invocations and returns.

Although the P space of a process may contain more than one segment, the state of a process may be saved into one controlling segment and then later restored. The ability to *freeze* a process into one segment and remotely activate that segment (and all necessary segments after that) using DSM provides *Clouds v.2* with a simple, easy way of performing process migration.

8.3 *Ra* Transport Protocol

The *Ra* Transport Protocol (RaTP) provides reliable message transactions over the Ethernet [Wil89]. The protocol is designed to be connectionless and is efficient for providing the request-reply form of communication that is common in client-server interactions. Since Clouds supports object invocations using RPC or DSM, this is the type of communication that is encountered in the system.

RaTP has been implemented both on *Ra* as well as on Unix. In addition to message transaction, RaTP provides interfaces to the RPC and DSM mechanisms. We are currently using RaTP to run the DSM clients on *Ra* and DSM servers on Unix file servers.

8.4 *Clouds* I/O System

The *Clouds* I/O system allows user objects to perform user-level terminal I/O. Note that *Clouds* does not support nor does it need user-level disk or network I/O. The I/O system is handled by a system object. Each user object is provided with two special sysnames of I/O objects (called *stdin* and *stdout*, inspired by Unix). These objects do not actually exist. They are operating system level pseudo-objects that support read and write calls. If a user object calls the write routine, the output reaches the (logical) terminal associated with the thread. Each logical terminal is a "text window" on a Unix workstation.

When a thread is created, a logical terminal is associated with the thread. The thread carries with it the sysname associated with this text window. Thus, all I/O calls made from object programs reach the user, regardless of where the thread is executing. I/O redirection can be done by simply changing the stdin/stdout sysnames to those of a user object that supports read and write calls. Further I/O will cause that user object to be invoked instead of the *Clouds* I/O system object.

8.5 Other Services

The other system objects used in the current implementation of *Clouds* include a Virtual Memory Manager, System Monitor, Buffer Manager, Ethernet driver, network disk partition, and DSM partition.

Devices in *Ra* conform to a standard interface: a class definition called *RaDevice* which is used to define device drivers. All device drivers must provide at least the methods defined in the base class *RaDevice*. They are *open(), close(), read(), write(), getmsg(), putmsg(), poll()* and *ioctl()*.

The system monitor provides a low-level monitor/shell capability. The monitor can be used to read and alter values of variables in the kernel or system objects, as well as execute arbitrary methods in the kernel or any system object. The system monitor can also be used for invoking user objects, thus providing a rudimentary shell for *Clouds*.

The network disk partition is implemented as a system object. Segments managed by this partition seem to reside on the *Clouds* node but actually reside on a Unix file server. Reads, writes, and control messages are shipped to the Unix system where a server operates on the Unix files that correspond to the indicated segments (see figure 8).

Many services are provided by Unix programs. These include terminal emulators, which run under Sunwindows and on top of RaTP interfacing through the user I/O system on *Ra*. The compiler is a modified Gnu C++ compiler that generates *Clouds* segments. The *Clouds* shell is a Unix program that is used to invoke *Clouds* objects using the RPC facility.

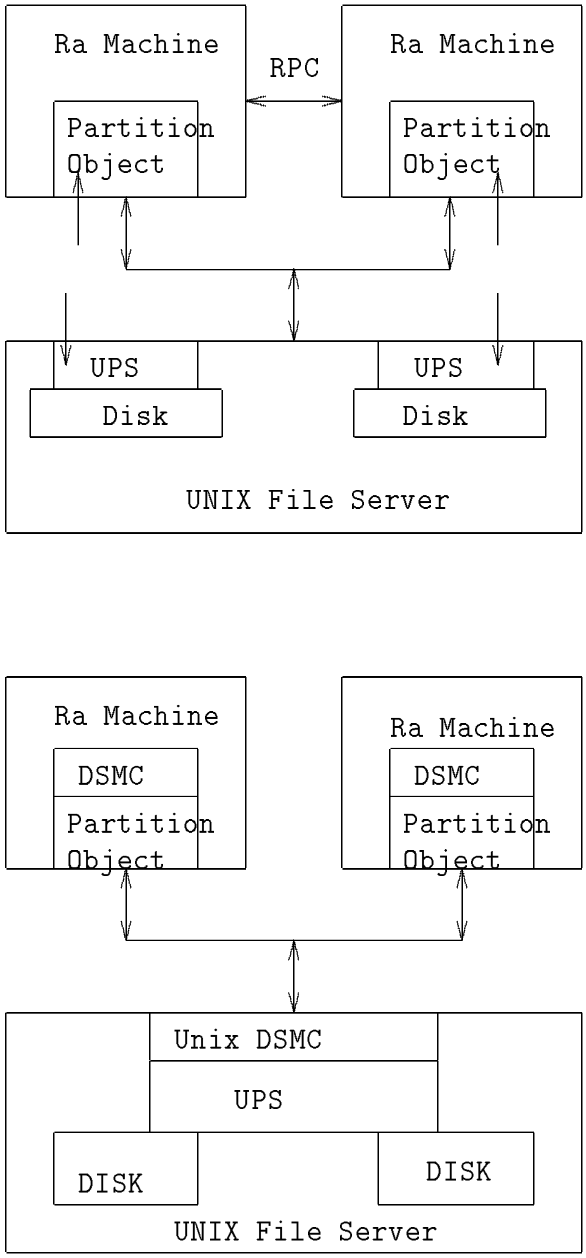
8.6 Status and Work in Progress

The facilities described above are operational. Project work is currently focusing on better system-level and user-level support tools for effective use of the *Clouds* operating system. This includes a user programming environment, a user-level naming system, and development of application programming techniques in the persistent object framework.

In addition, work is being done in the area of reliability and persistent-object programming, the thrust of the original *Clouds* system. We have explored memory semantics for programming persistent objects and have developed a set of flexible mechanisms that support customized data consistency [CD89, DC90, Che90]. In other fault tolerance research, we have developed a scheme that replicates data as well as computation to guarantee forward progress of computations [ADL90]. Work is underway to design schemes that exploit multicast communication to make a variety of services (e.g. object location, group communication, commit protocols, replication management and so on) more efficient [AB89, BA89].

9 Concluding Remarks

Clouds is intended to serve as a base for research in distributed computing at Georgia Tech. The new *Clouds* kernel, *Ra*, coupled with system objects, provides an elegant environment for operating systems development. The design and implementation of *Ra* benefited from the experience gained from the design and implementation of the first *Clouds* kernel.



UPS = Unix Partition Server

Figure 8: Network Disk and DSM Services

The design and implementation of *Clouds v.2* are geared more towards flexibility, portability, and maintenance than the *Clouds v.1* kernel. This is reflected in the design of the *Ra* kernel and its virtual memory architecture, support for lightweight kernel daemons, and system object facility. Furthermore, the additional freedom allowed by *Ra* facilitates more easy testing of alternative system designs (both mechanisms and algorithms) using *Ra* as the implementation base.

We would like to acknowledge all those who have worked on the *Clouds* project, past and present, for making this design and implementation possible. This includes Jim Allchin, Martin McKendry, Gene Spafford, Dave Pitts, Tom Wilkes and Henry Strickland for their contributions to *Clouds v.1*, and Nasr Belkeir, M. Chelliah, Vibby Gottemukkala, Ranjit John, Ajay Mohindra, Gautam Shah, and Monica Skidmore for their contributions to *Clouds v.2*.

References

[AB89] M. Ahamad and N. E. Belkeir. Using Multicast Communication for Dynamic Load Balancing in Local Area Networks. In *14th Annual Conf. on Local Computer networks*, October 1989.

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for Unix Development. In *Proceedings Summer Usenix*, July 1986.

[ADL90] M. Ahamad, P. Dasgupta, and R. J. LeBlanc. Fault-Tolerant Atomic Computations in an Object-based Distributed System. *Distributed Computing Journal*, April 1990.

[All83] J. E. Allchin. *An Architecture for Reliable Decentralized Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1983. (Available as Technical Report GIT-ICS-83/23).

[BA89] N. Belkeir and M. Ahamad. Low Cost Algorithms for Message Delivery in Dynamic Multicast Groups. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 1989.

[BHK+88] J. M. Bernabéu Aubán, P. W. Hutto, M. Y. 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. In *European UNIX systems User Group Autumn Conference*, October 1988.

[BHK+89] J. M. Bernabéu Aubán, P. W. Hutto, M. Y. Khalidi, M. Ahamad, W. F. Appelbe, P. Dasgupta, R. J. LeBlanc, and U. Ramachandran. The Architecture of *Ra*: A Kernel for *Clouds*. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, January 1989.

[CD89] R. C. Chen and P. Dasgupta. Linking Consistency with Object/Thread Semantics: An Approach to Robust Computation. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 1989.

[Che90] Raymond C. Chen. *Consistency Mechanisms and Memory Semantics for Persistent Object-Based Distributed Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1990. (In Progress.).

[CJR87] R. Campbell, G. Johnston, and V. Russo. Choices (Class Hierarchichal Open Interface for Custom Embedded Systems). *Operating System Review*, 21(3), July 1987.

[CZ83] D. Cheriton and W. Zwaenpoel. The Distributed V Kernel and its Performance for Diskless Workstations. In *Proceedings Of the 9th ACM Symposium on Operating System Principles. (Breton Woods, N.H.)*, pages 129–139, October 10–13 1983.

[DC90] P. Dasgupta and R. C. Chen. Memory Semantics for Programming Persistent Objects. (In Progress.), 1990.

[Ken86] Greg Kenley. An Action Management System for a Distributed Operating System. Master's thesis, School of Information and Computer Science, Georgia Institute of Technology, 1986.

[Kha89] M. Yousef A. Khalidi. *Hardware Support for Distributed Object-based Systems.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, June 1989. (Available as Technical Report GIT-ICS-89/19).

[Mos81] J. Moss. Nested transactions: An approach to reliable distributed computing. Technical Report MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[NKK86] E. Nett, J. Kaiser, and R. Kroger. Providing Recoverability in a Transaction Oriented Distributed Operating System. In *6th International Conference on Distributed Computing Systems*, pages 590–597, May 1986.

[Nor87] J. D. Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems*, volume 16 of *Perspectives in Computing.* Academic Press, 1987. Editors: W. Rheinboldt and D. Siewiorek.

[Org72] E. E. Organick. *The Multics System.* MIT Press, 1972.

[Pit86] David V. Pitts. *A Storage Management System for a Reliable Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1986. (Available as Technical Report GIT-ICS-86/21).

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer. In *Eighteenth Annual International Conference on Parallel Processing*, August 1989.

[RR81] R. Rashid and G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64–75, December 1981.

[RTY+87] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 31–39, Palo Alto, CA, October 1987.

[Spa86] Eugene. H. Spafford. *Kernel Structures for a Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Tech, Atlanta, Ga, 1986. (Available as Technical Report GIT-ICS-86/16).

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 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. *Communications of the ACM*, 17(6):337–345, June 1974.

[Wil87] C. Thomas Wilkes. *Programming Methodologies for Resilience and Availability.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987. (Available as Technical Report GIT-ICS-87/32).

[Wil89] Christopher J. Wilkenloh. Design of a Reliable Message Transaction Protocol. Master's thesis, School of Information and Computer Science, Georgia Institute of Technology, 1989.

[WLH81] W. A. Wulf, R. Levin, and S. P. Harbison. *HYDRA/C.mmp, An Experimental Computer System*. McGraw-Hill, Inc., 1981.