The Clouds Distributed Operating System *

Partha Dasgupta

Dept. of Computer Science and Engg.
Arizona State University, Tempe AZ 85287-5406.

Richard J. LeBlanc Jr.,
Mustaque Ahamad and Umakishore Ramachandran.

College of Computing
Georgia Tech, Atlanta, GA 30332

Keywords: Distributed Operating Systems, Object-based Systems, Persistent Object Systems and Fault-tolerance.

Abstract

Clouds is a distributed operating system that runs on general purpose computers connected via a local-area network. The system is composed of compute servers, data servers and user workstations. It implements an object-thread model of computation that is based on the object-oriented programming concepts. Clouds supports coarse-grained objects that provide long term storage for persistent data and associated code. Lightweight, concurrent threads provide support for computational activity through the code in the objects.

Persistent objects and threads give rise to a programming environment composed of shared permanent memory, dispensing with the need for facilities such as file systems. Though the hardware may be distributed, Clouds provides applications with a logically centralized system, based on a shared "single-level" store.

The implementation of Clouds separates operating system policies from the mechanisms used to implement them. The structure of Clouds is a three-level hierarchy. The lowest level is a minimal kernel that provides mechanisms for memory and processor control. At the next level, a set of trusted system-objects provide low-level operating system services. High-level services are implemented at the top level in application objects.

Clouds is a native operating system, that is it is not hosted on some other operating system. It runs on a set of Sun-3 machines connected by an Ethernet. It is being used for conducting research in distributed systems. This paper describes the Clouds object-thread model, programming environment and usage details. The implementation overview and a synopsis of the research being conducted using Clouds is also included.

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

1. Introduction

A distributed operating system is a control program running on a set of computers that are interconnected by a network. This control program unifies the different computers into a single integrated compute and storage resource. Depending on the facilities provided by the system, such systems get classified as general purpose, real time, or embedded systems.

The need for distributed operating systems stems from the rapid change in the hardware environment in many organizations. The proliferation of workstations, personal computers, data and compute servers, and networking in the last decade (due to the rapidly falling prices of hardware) has underlined the need for efficient and transparent management of these physically distributed resources (see box).

This paper presents a paradigm for structuring distributed operating systems, its potential and implications for users, and research directions for the future.

Distributed Operating Systems

Computing environments composed of networked computers are now commonplace. Due to the affordable prices of powerful desktop systems, most computing environments are now composed of combinations of workstations and file servers. Such distributed environments however are not easy to use or administer.

Using a collection of computers connected by a local-area network often poses problems of resource sharing and environment integration not present in centralized systems. To keep user productivity high, it is necessary to make the distribution transparent and make the environment appear to be centralized. The short-term solution adopted by current commercial software is to extend conventional operating systems to allow transparent file access and sharing. For example, Sun added Network File System (NFS) to provide distributed file access capabilities in Unix. Sun-NFS has become the industry standard distributed file system for Unix systems. The small systems world dominated by IBM-PC compatible and Apple computers has software packages that perform multitasking and network-transparent file access. Such facilities are provided by packages such as Microsoft Windows 3.0, Novell Netware, Appletalk, and PC-NFS.

A better long-term solution is the design of an operating system that takes the distributed nature of the hardware architecture into consideration at all levels. Such an operating system, for distributed hardware, is a *distributed operating system*. A distributed operating system makes a collection of computers look and feel like one centralized system, yet keeps intact the advantages of distribution. Message-based and object-based systems are two paradigms for structuring such operating systems.

1.1. Distributed Operating Systems

Operating system structures for a distributed environment follow one of two major paradigms: *Message-based* or *Object-based*. Message-based operating systems place a *message passing kernel* on each node, supporting processes and communication between them via explicit messages. This kernel supports both *local communication* (communication between processes on the same node) and *non-local* or *remote* communication, sometimes implemented via a distinguished network manager process. In a traditional system such as Unix, access to system services is requested via *protected procedure calls*, whereas in a message-based operating system it is requested via *message passing*. Message-based operating systems are attractive for structuring distributed systems due to the separation of policy, encoded in server processes, from mechanism implemented in the kernel.

Object-based distributed operating systems encapsulates services and resources into entities called objects. Objects are similar to instances of abstract data types, and are written as individual modules composed of the specific operations that define their interface. Access to system services is requested by invoking the appropriate system object. The invocation mechanism is similar to a protected procedure call. Objects encapsulate functionality, similar to the encapsulation provided by server processes in message-based systems.

Among the well-known message-based systems are the V-system [Ch88] developed at Stanford and the Amoeba system [Mu*90] developed at Vrije University. These systems provide computation and data services via servers that run on machines linked by a network.

Most of the object-based systems are built on top of an existing operating system, typically Unix. Examples of such systems include Argus [Lis87], Cronus [Be*85], and Eden [Al*85]. These systems support objects that respond to invocations sent to them via the message-passing mechanisms of Unix.

Mach is a recent operating system that has a distinctive character [Ac*86]. Mach is a Unix compatible operating system built in a machine independent fashion and it runs on a large variety of uniprocessors and multiprocessors. It has a small kernel handling the virtual memory and process scheduling with other services built on top of the kernel. Mach implements mechanisms that provide for distribution, especially through a facility called memory objects. Memory objects provide a means for sharing memory between separate tasks executing on possibly different machines.

1.2. The Clouds Approach

Clouds is a distributed operating system that integrates a set of nodes into a conceptually centralized system. The system is composed of *compute servers*, *data servers* and *user workstations*. A compute server is a machine that is available for use as a computational engine; a data server is a machine whose purpose is to function as a repository for long-lived (i.e., persistent) data. A user workstation is a machine whose sole purpose is to provide the programming environment for the user to develop applications and interface with the compute and data servers for executing these applications on them. Note that if a disk is

associated with a compute server, it can also serve as a data server for other compute servers. Clouds is a native operating system, that runs on top of a native kernel called "*Ra*"*. It currently runs on Sun-3/50 and Sun-3/60 computers and cooperates with Sun SparcStations running Unix that provide user interfaces.

Clouds is a general purpose operating system. That is, it is intended to support all types of languages and applications, distributed or not. All applications can view the system as a monolith, but distributed applications may choose to view the system as composed of several separate compute and data servers. Each compute facility in Clouds has access to all resources in the system.

The system structure is based on an object-thread model. The object-thread model is an adaptation of the popular *object-oriented programming* model which structures a software system as a set of *objects*. In the object-oriented model, Each object is an instance of an abstract data type consisting of data and operations on the data. The operations are called *methods*. An object is an instance of a *class* that defines the type of the object. A class may have any number of (0 or more) instances, but an instance is derived from exactly one class. Objects respond to *messages*. Sending a message to an object causes the object to execute a method. The execution of a method accesses or updates data stored in the object and may cause messages to be sent to other objects. Upon completion, the method sends a reply to the sender of the message.

Clouds has a similar structure, implemented at the operating system level. Clouds objects are large-grained encapsulations of code and data that are contained in an entire virtual address space. An object is an instance of a class, and a class is a compiled program module. Clouds objects respond to *invocations*. An invocation is the result of a thread of execution entering the object to execute an operation (or method) in the object.

Clouds provides objects to support an abstraction of storage and threads to implement computations. This decouples computation and storage, thus maintaining their orthogonality. In addition, the object-thread model unifies the treatment of I/O, inter-process communication, information sharing and long-term storage. This model has been further augmented to support atomicity and to provide support for reliable execution of computations.

Multics was the starting point for many ideas found in the operating systems of today, and Clouds is no exception. These include sharable memory segments and single level stores using mapped files. Using objects as a system structuring concept was first implemented in Hydra [Wu*74]. Hydra ran on a multiprocessor and provided named objects for OS services.

2. The Clouds Paradigm

This section elaborates on the object-thread paradigm of Clouds, illustrating the paradigm with examples of its usage.

---

* After the Egyptian Sun-God.

What can objects do?

Conceptually, an object is an encapsulation of data and a set of operations on the data. The operations are performed by invoking the object, and can range from simple data manipulation routine to complex algorithms; from shared library accesses to elaborate system services.

Objects can also provide specialized services. For example, a sensing device can be represented as an object and an invocation can be used to gather data from the device, without having to know about the mechanisms involved in accessing the device, or even the location of the device. Similarly, terminal I/O can be effectively handled by an object (with read and write operations defined on it).

Objects can be active. An active object has one or more processes associated with it that communicate with the external world and handle housekeeping chores internal to the object. For example a process may monitor the environment of the object and may inform some other entity (another object) on the occurrence of an event. This feature is particularly useful in objects that manage sensor monitoring devices.

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

2.1. Objects

A Clouds object is a persistent (or non-volatile) virtual address space. Unlike virtual address spaces in conventional operating systems, the contents of a Clouds object are long-lived. That is, 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". They are best suited for storage and execution of large-grained data and programs because of the overhead associated with invocation and storage of objects.

Unlike objects in some object-based operating systems, a Clouds object does not contain a process (or thread). Thus Clouds objects are passive. Since contents of a virtual address space are not accessible from outside the ad-

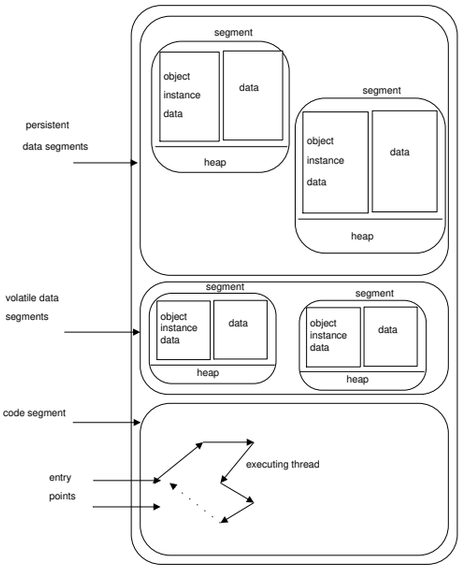

Figure 1: A Clouds Object.

dress space, the memory (data) in an object is accessible only by the code in the object.

A Clouds object contains user defined code, persistent data, a volatile heap for temporary memory allocation, and a persistent heap for allocating memory that becomes a part of the persistent data structures in the object (Figure 1). Recall that the data in the object can be manipulated only from within the object. Data can be passed into the object when an entry point is invoked (input parameters). Data can be passed out of the object when this invocation terminates (result parameters).

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. Therefore, the sysname-based naming scheme in Clouds creates a uniform, flat system name space for objects. Users can define high-level names for objects. These are translated to sysnames using a name server. Objects are physically stored in data servers, but are accessible from all compute servers in the system (see section 3.2), thus providing location transparency to the users.

2.2. Threads

The only form of user activity in the Clouds system is the user thread. A thread is a logical path of execution that executes code in objects, traversing objects as it executes. Thus unlike a process in a conventional operating system, a Clouds thread is not bound to a single address space. A thread is created by an interactive user or under program control. When a thread executes an entry point in an object, it accesses or updates the persistent data stored in it. In addition, the code in the object may invoke operations in other objects. In such an event, the thread 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 completes and returns results. These arguments/results are strictly data; they may not be addresses. This restriction is mandatory as addresses in one object are meaningless in the context of another object. In addition, object invocations can be nested or recursive. After the thread completes execution of the operation it was created to execute, it terminates.

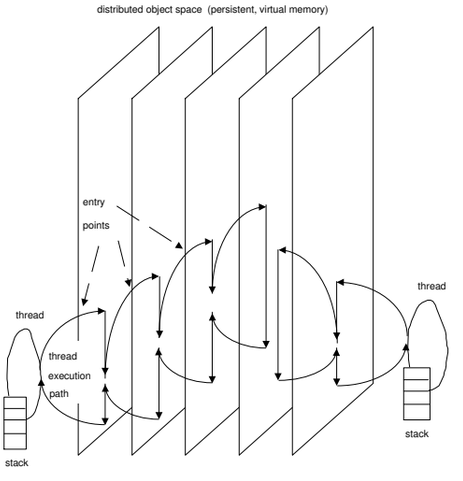

Figure 2: Distributed Object Memory.

Due to the nature of Clouds objects, a thread cannot access any data outside the current address space (object) in which it is executing. Control transfer between address spaces occurs through object invocation and data transfer between address spaces occurs through parameter passing.

Several threads can simultaneously enter an object and execute concurrently. Multiple threads executing in the same object share the contents of the object's address space. Figure 2 shows thread ex-

No Files? No Messages?

The persistent objects supported in an operating system like Clouds provide a structured permanent storage mechanism that can be used for a variety of purposes including the simulation of files and messages. Data in any form can be stored in an object and invocations can be used to:

- Manipulate or process the stored data.
- Ship data in and out of the object in forms not necessarily the same as used for storage.
- Allow controlled concurrent access to shared data, without regard to the location of the data.

In a persistent programming environment, there is no need for files. Files are used as byte sequential storage of long-lived data in conventional systems. When persistent shared memory is available there is no need to convert data into byte-sequential form and store them in files (and later retrieve and reconvert). The data can be kept in memory, in a form controlled by the programs (e.g. lists, trees), even when not in use.

In fact, files can be simulated by objects that store byte sequential data and have read and write invocations defined to access this data. Such an object will look like a file, even though the operating system does not explicitly support files.

The same is true for messages. The duality of messages and shared memory is well known. If desired, a buffer object with the send and receive invocations defined on it can serve as a port structure between two (or more) communicating processes.

We feel that files, messages and disk I/O are artifacts of the way hardware is structured. Given an object implementation, these features are neither necessary nor attractive. In fact, new programming paradigms based on object-oriented styles are emerging, which use persistent memory effectively, and do not use files and messages.

ecutions in the Clouds object spaces. Concurrency control within the object is handled by the programmer of objects using system supported synchronization primitives such as locks or semaphores.

2.3. The Interaction Between Objects and Threads

The structure created by a system, composed of objects and threads, has several interesting properties. Inter-object interfaces are procedural. Object invocations are equivalent to procedure calls on long-lived modules that do not share global data. The invocations work across machine boundaries.

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 for programs and data, and is volatile, i.e., 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 simpler. Persistent objects provide a structured single-level store which is cosmetically similar to mapped files in Multics and SunOS.

Some systems use message-passing for communicating shared data and coordinating computations. Sharing of data in Clouds is achieved by placing the shared data in an object. Computations that need access to shared data invoke the object where the data exists. Messages and files are not supported at the operating system level. They can be simulated by objects if necessary (see box).

In a message-based system, the user has to determine the desired level of concurrency at the time of writing an application. This is programmed as a certain number of server processes. The object-thread model of Clouds eliminates the need for determining the extent of concurrency at the time of writing the application. An object can be written from the point of view of the functionality that it is meant to provide, than the actual level of concurrency that it may have to support. At execution time, the level of concurrency can be specified by creating concurrent threads to execute in the objects that comprise the user level application. The application objects, however, have to be written to support concurrent executions, using synchronization primitives such as semaphores and locks.

To summarize:

- The Clouds system is composed of named address spaces called objects. Objects provide data storage, data manipulation, data sharing, concurrency control, and synchronization.
- Control flow is achieved by threads invoking objects.
- Data flow is achieved by parameter passing.

2.4. Programming in the Clouds Model

To the programmer, there are two kinds of Clouds objects: classes and instances. A class is a template that is used to generate instances. An instance is an object that is invocable by user threads. Thus to write application programs for Clouds, a programmer writes one or more Clouds classes that define the code and data of the application. The programmer may then create the requisite number of instances of these classes. The application is then executed by creating a thread to execute the top-level invocation that runs the application.

To give the reader a flavor of programming in the Clouds system, the following simple example is presented. The object "**`rectangle`**" consists of **`x`** and **`y`** dimensions of a rectangle. The object has two entry points, one for setting the size of the rectangle and the other for computing the area. The object is defined as follows:

```
clouds_class rectangle;
int x, y;              // persistent data for rect.
entry rectangle;       // constructor
entry size (int x, y); // set size of rect.
entry int area ();     // return area of rect.
end_class
```

Once the class is compiled, any number of instances may be created either from the command line or via another object. Suppose the rectangle class is instantiated into an object called "**Rect01**". Now **Rect01.size** can be used to set the size and **Rect01.area** can be called to return the area of the rectangle. The entry point in the object may be called by a command in the command interpreter for Clouds. Entry points may also be invoked in the program, allowing one object to call another.

Objects have user names, which are assigned by the programmer when objects are created (compiled or instantiated). The sysname of an object is then obtained by using a name server that translates the user name to the sysname. Recall that a sysname is a unique name for an object, which is needed for invoking an object. The following code fragment details the steps in getting access to a Clouds object **Rect01** and invoking operations on it:

```
rectangle_ref  rect;   // "rect" is a class that refers to
                       // an object of type rectangle.

rect.bind("Rect01");          // call to name server,
                              // binds sysname to Rect01
rect.size(5, 10);             // invocation of Rect01
printf("%d\n" rect.area() );  // will print 50
```

Clouds provides a variety of mechanisms including registering user-defined names of objects with the name server, looking up names using the name server, invoking objects both synchronously and asynchronously, and synchronizing threads that share data. In the interest of space, these details are not discussed in this paper.

I/O to the user console is handled by read and write routines (and printf, scanf library calls). These routines read/write ASCII strings to and from the user terminal, irrespective of the actual location of the object or the thread, as long as the thread was started at the user terminal.

User objects and their entry points are typed by the language definition. Static type checking is performed on the object and entry point types at compile time. No runtime type checking is done by Clouds. Clouds objects are coarse-grained, unlike fine-grained entities found in object-oriented programming languages such as Smalltalk. Since an object invocation in Clouds is at least an order of magnitude more expensive than a simple procedure call, a Clouds object is appropriate to be used as a module that may contain

several fine-grained entities. These fine grained objects are completely contained within the Clouds object and are not visible to the operating system.

Currently we support two languages in the Clouds operating system. CC++ is an extension of C++ that is used by systems programmers. Distributed Eiffel is an extension of Eiffel that is targeted for application developers. Both CC++ and Distributed Eiffel have been designed to support persistent fine-grained and large-grained objects, invocations, thread creation, synchronization and user-level object naming.

3. The Clouds Environment

The Clouds system integrates a set of homogeneous machines into one seamless environment that behaves as one, large computer. The system configuration is composed of three logical categories of machines, each supporting a different logical function. These are *compute servers, data servers* and *user workstations*.

The core of the system consists of a set of homogeneous machines of the compute server category. Compute servers do not have any secondary storage. These machines provide an execution service for threads. Secondary storage is provided by data servers. Data servers are used to store Clouds objects and supply the code and data of these objects to compute servers. The data servers also provide support for distributed synchronization. The third machine category is the user workstation. These machines provide user access to Clouds compute servers. Compute servers, in turn, know how and when to access data servers.

The logical machine categories do not have to be mapped to physical machines using a one-to-one scheme. Although a diskless machine can function only as a compute server, a machine with a disk can simultaneously be a compute and data server. This enhances computing performance, since data access via local disk is faster than data access over a network. However, in our prototype system, we use a one-to-one mapping, in order to keep the system's implementation and configuration simpler (Figure 3).

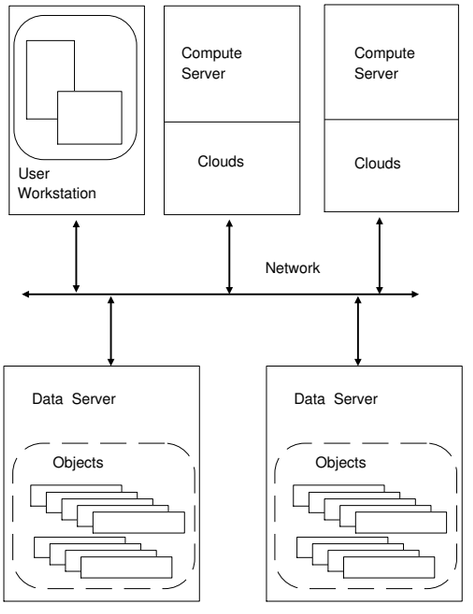

Figure 3: The Clouds System Architecture

The user interface to Clouds is provided by a suite of programs that run on top of Unix on Sun workstations. These programs include Distributed Eiffel and CC++ compilers, Clouds user shell (under X-windows), a user-I/O manager and various utilities. The user can use the familiar Unix utilities (including Unix editors) to interface with these programs.

3.1. The User Environment

A user writes Clouds programs using CC++ or Distributed Eiffel and compiles them on the Unix workstation. The compiler loads the

generated classes on a Clouds data server. Now these classes are available to all Clouds compute servers. Any Clouds node (or a user on a Unix machine) can create instances of these classes and generate invocations to the objects thus created. Note that the objects once created become part of the Clouds persistent object memory and can be invoked until they are explicitly deleted.

A user invokes a Clouds object by specifying the object, the entry point and the arguments to the Clouds shell. The Clouds shell sends an invocation request to a compute server and the invocation proceeds under Clouds using a Clouds thread. The user communicates to the thread via a terminal window in the X-window system. All output generated by the thread (regardless of where it is executing) appears on the user terminal window and input to the thread is provided by typing in the window.

3.2. The System Environment

As we mentioned earlier, the hardware environment consists of compute servers and data servers with some nodes providing both functions. Starting a user level computation on Clouds involves first selecting a compute server to execute the thread. This is a scheduling decision and may depend on such factors as scheduling policies and the load at each compute server, and availability of the resources needed for the computation. Once this decision has been made, the second task is bringing the object in which the thread has to execute from the data server to the compute server. This requires a remote paging facility;

Distributed Shared Memory

The name space represented by the Clouds objects constitutes a shared sparse address space. Since each object itself consists of a linear address space, these two spaces in conjunction provide a system-wide 2-dimensional address space. The contents of this address space are available on every machine in the system, providing a globally shared (yet distributed) memory.

The sharing of this global memory is provided by a mechanism that we call DSM (Distributed Shared Memory). When a thread on node ***A*** invokes an operation on object ***O***, the invocation gets executed on node ***A***. If ***O*** is not located on ***A***, this causes a series of page faults which are serviced by demand paging the pages of ***O*** from the data server(s) where they currently reside. Thus, only the necessary parts of the code and data of ***O*** are brought to ***A***.

However if ***O*** is being used at both node ***A*** and node ***B***, care must be taken to ensure that at all times ***A*** and ***B*** see the exact same contents of ***O***. This is called one-copy semantics. The maintenance of one-copy semantics is achieved by coherence protocols that are an integral part of the DSM access strategy.

The implication of this mechanism is that every object on the system *logically* resides at every node. This is a powerful concept that separates object storage from its usage, effectively exploiting the physical nature of the distributed system which is composed of compute servers and data servers.

coupled with this requirement is the fact that all objects are potentially shared in the Clouds model. Therefore the entity that provides the remote paging facility should be cognizant of the need to provide a way of maintaining the consistency of shared pages. In Clouds this is satisfied by a mechanism called *distributed shared memory* (DSM), which supports the notion of shared memory on a non-shared memory (distributed) architecture (see box). The data servers execute a coherence protocol that preserves single-copy semantics for all the objects [Li*89]. With DSM, concurrent invocation of the same object by threads at different compute servers is possible. Such a scenario would result in multiple copies of the same object existing at more than one compute server with DSM providing the consistency maintenance.

Suppose an thread is created on compute server ***A*** to invoke object O_1. The compute server retrieves a header for the object from the appropriate data server, sets up the object space and starts the thread in that space. As the thread executes in the object space, the code and data of the object that is accessed by the thread is demand paged from the data servers (possibly over the network).

If the thread executing in O_1 generates an invocation to object O_2, the system may choose to execute the invocation on either ***A*** itself or on a different compute server ***B***. In the former case, if the required pages of object O_2 are at other nodes, they have to be brought to node ***A*** using DSM. Once the object has been brought into ***A***, the invocation proceeds the same way as when O_1 resides at ***A***. On the other hand, the system may choose to execute the invocation on a different compute server ***B***. In this case the thread sends an invocation request to ***B***, which invokes the object O_2 and returns the results to the thread at ***A***. This scenario is similar to the remote procedure call (RPC) found in other systems such as the V system but is more general because ***B*** does not have to be the node where O_2 currently resides.

The compute and data server scheme makes all objects accessible to all compute servers. The DSM coherence protocol ensures that the data in an object is seen by concurrent threads in a consistent fashion even if they are executing on different compute servers. The synchronization support provided by data servers allows threads to synchronize their actions regardless of where they execute.

4. The Implementation of Clouds

The implementation of Clouds uses a *minimalist* approach towards operating system development. With this approach, each level of the implementation consists of only those functions that cannot be implemented without a significant performance penalty at a higher level. Traditional systems such as Unix provide most of the operating system services in one big monolithic kernel. Unlike such systems, we differentiate between the kernel of the operating system and the operating system itself. This approach makes the system modular, easy to understand, more portable, and convenient to enhance. High-level features can be implemented as user-level libraries, objects or services that use the low-level mechanisms in the operating system. Further, it provides a clean separation of policy from mechanisms; that is the policies are implemented at the high-level using the mechanisms at the lower-level.

The current implementation of Clouds consists of three levels: at the lowest level is Ra which provides the mechanisms for managing the basic resources, namely, processor and memory. The next level up is a set of *system objects* which are trusted software modules providing essential system services. Finally, other non-critical services such as naming and spooling are implemented as user objects to complete the functionality of Clouds.

4.1. The Ra Kernel

Ra is the native minimal kernel that supports the basic mechanisms: virtual memory management and low-level scheduling. Ra implements the following abstractions.

- *Segments:*
  A segment is a sequence of uninterpreted bytes of variable length that exists either on the disk or in physical memory. Segments have systemwide unique names (called sysnames). Segments once created, persist until explicitly destroyed.

- *Virtual Spaces:*
  A virtual space is the abstraction of an addressing domain, and is a monotonically increasing range of virtual addresses with possible holes in the range. Each contiguous range of virtual addresses is mapped to (a portion of) a segment.

- *IsiBas:*
  An *IsiBa*[*] is the abstraction of activity in the system, and can be thought of as a light-weight process. It is simply a kernel resource that should be associated with a stack to realize a schedulable entity. There are several types of stacks in the system (e.g. kernel, interrupt and user), and an IsiBa can use an instance of any type of stack. A Clouds process is an IsiBa in conjunction with a user stack and a Ra virtual space. One or more Clouds processes are used to build a Clouds thread. IsiBas can also be used for a variety of purposes inside system objects, including interrupt services, event notification, and watchdogs.

- *Partitions:*
  A partition is an entity that provides non-volatile data storage for segments. A Clouds compute server may have one or more partitions, but a segment belongs to exactly one partition. In order to access a segment, the partition containing the segment has to be contacted. The partition communicates with the data server where the segment is stored to page the segment in and out when necessary. Note that Ra only defines the interface to the partitions. The partitions themselves are implemented as system objects which are discussed in section 4.2.

The relationship between segments, virtual spaces and partitions are depicted in Figure 4.

The implementation of Ra is separated into machine-dependent and machine-independent parts. All components of Ra are built using the class mechanisms of C++. The scheme of

---

* From Ancient Egyptian: "Isi" = light, "Ba" = soul.

using system objects enhances the object structure of Ra. Ra consists of 6000 lines of machine dependent C++ code, 6000 lines of machine independent C++ code and 1000 lines of Sun (68020) assembly code. It currently runs on the Sun-3 class machines. More details about the implementation are in [Da*90a].

4.2. System Objects

Ra can be thought of as the conceptual *motherboard*. Operating system services are provided on top of Ra by *system objects*. System objects are independently compiled modules of code that have access to certain operations defined by Ra. These operations are exported as kernel classes by Ra and are inherited by the system objects. Conceptually, the system objects are similar to Clouds objects living in their own virtual space and supporting external invocations and having access to operations in the Ra kernel. However, for the sake of efficiency, system objects live in the kernel space, are linked to the Ra kernel at system configuration time, and are not directly invocable from user level. System objects are implicitly invoked through a system-call interface available to user-level objects.

Some system objects implement low-level functions inside the operating system, including the buffer manager, the uniform I/O interface, and the Ethernet driver. Other system objects implement high-level functions that are invoked indirectly as a result of a system call. These objects include the thread manager, the object manager and the user I/O manager.

The following paragraphs describe some of the important system objects.

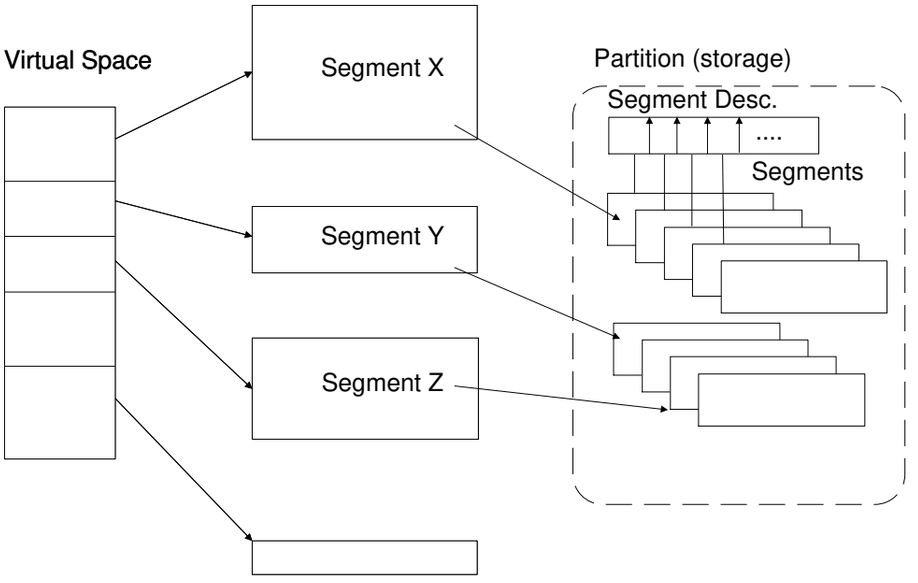

Figure 4: Virtual spaces, segments and partitions.

- Thread Manager:
  As we mentioned earlier, a thread may span machine boundaries and is implemented as a collection of Clouds processes. There is some information associated with a thread such as the objects it may have visited, the user workstation from which it was created, and the windows on the user workstation with which it has to communicate when input/output requests are made during the computation. The thread manager is responsible for the creation, termination, naming and all bookkeeping necessary to implement threads.

- User Object Manager:
  User-level objects are implemented through a system object called the object manager. The object manager creates and deletes objects and provides the object invocation facility. An object is stored in a Ra virtual space. The invocation of an object by a thread is handled mainly by the object manager in conjunction with the thread manager. Briefly, when a thread invokes an object, the stack of the thread invoking the object is mapped into the same virtual address space as the object and the thread is allowed to commence execution at the entry point of the object. When the execution of the operation terminates, the object manager unmaps the thread stack from the object and remaps it in the object where the thread was previously executing. If there was no previous object, then the object manager informs the thread manager and the thread is terminated.

- DSM Clients and Servers:
  DSM clients and servers are partitions that interact with the data servers to provide one-copy semantics for all object code and data that are used by the Clouds nodes. When a page of data is needed at node ***A***, the DSM client partition requests it from the data server. If the page is currently in use in exclusive mode at node ***B***, the data server forwards the request to the DSM server at node ***B***, which supplies the page to ***A***. The DSM server allows maintaining (both exclusive and shared) locks on segments and provides other synchronization support.

- User I/O Manager:
  This system object provides support for Clouds computations to read from and write to user terminals. A user terminal is a window on a Unix workstation. When a thread executes a write system call, the I/O manager routes the data written to the appropriate controlling terminal. Reads are handled similarly. The user I/O manager is a combination of a Ra system object and a server on each Unix workstation.

- Networking and RaTP:
  Networking is handled by two system objects: the Ethernet driver and the network protocol. The network protocol used for all communication in Clouds is a transport layer protocol called the Ra Transport Protocol (*RaTP*). It is similar to the communication protocol VMTP [Ch86], used in the V-system, and provides efficient, reliable connectionless message transactions. A message transaction is a send/reply pair used for client-server type communications. RaTP has been implemented both on Ra (as a system object) and on Unix, allowing Clouds to Unix communication.

4.3. Current Status and Performance

All the features mentioned in the paper thus far have been implemented and are in use. The compute servers run on diskless Sun-3/60 machines; data servers and user workstations are Sun SPARCstation-1 machines running UNIX. The data service is done by storing the data in Unix files and the user workstations run X-Windows. The user interface includes the Clouds terminal, Clouds shell, the CC++ compiler and the Distributed Eiffel compiler. All communication between nodes running Clouds or Unix use the RaTP protocol.

The kernel performance is good. Context switch time is 0.14 ms. The time to service a page fault when the page is resident on the same node costs 1.5 ms for a zero-filled, 8K page; and costs 0.629 ms for a non zero-filled page.

Networking is one of the most heavily used subsystem of Clouds, especially since our current implementation uses diskless compute servers. All objects are demand paged to the servers over the network when used. The RaTP protocol handles the reliable data transfer between all machines. The Ethernet round-trip time is 2.4 ms; this involves sending and receiving a short message (72 bytes) between two compute servers. The RaTP reliable round-trip time is 4.8 ms. To reliably transfer an 8K page from one machine to another costs 11.9 ms, compared to 70 ms using Unix FTP and 50 ms using Unix NFS.

Object invocation costs vary widely depending upon whether the object is currently in memory or have to be fetched from a data server. The maximum cost for a null invocation is 103 ms while the minimum cost is 8 ms. Note that due to locality the average costs is much closer to the minimum than the maximum.

5. Using Clouds for Distributed Systems Research

This section presents a brief overview of some topics being investigated as part of the continuing systems research in the Clouds project

5.1. Using Persistent Objects

Persistent shared single-level storage is the central theme of the Clouds model. Therefore, effectively supporting and exploiting persistent memory in a distributed setting has been the thrust of several related research projects. Another area of research is in harnessing the distributed resources to realize speedups for executing specific applications compared to a single-processor implementation. Some of these projects are summarized below.

- *Distributed Programming*:
  Using the DSM feature of Clouds, centralized algorithms can be run as distributed computations with the expectation of achieving speedup. For example, sorting algorithms can use multiple threads to perform a sort, with each thread being executed at a different compute server, even though the data itself is contained in *one* object. The threads work on the data in parallel and those parts of the data that are in use at a node migrate to that node automatically. We have shown that even though the data resides in a single object, the computation can be run in a distributed fashion without incurring

a high overhead. These experiments are helping us understand the trade-off between computation and communication, and the granularity of computations that warrant distribution.

- *Types of Persistent Memory*:
  Persistent memory needs a structured way of specifying attributes such as longevity and accessibility for the language-level objects contained in Clouds objects. To this end we provide several types of memory in objects. The sharable, persistent memory is called per-object memory. We also provide per-invocation memory that is not shared, yet is global to the routines in the object and lasts for the length of each invocation. Similarly per-thread memory is global to the routines in the object but specific to a particular thread and lasts until the thread terminates. Such a variety of memory structures provides a powerful programming support in the Clouds system [Da*90b].

- *Lisp Programming Environment*:
  If the address space containing a Lisp environment can be made persistent, it has several advantages, including not having to save/load the environment on startup and shutdown. Further, by invoking entry points in remote Lisp interpreters it is possible to allow inter-environment operations that are useful in building knowledge-bases. Other features that naturally arise due to the distributed nature of the system include concurrent evaluations and load sharing.

- *Object-Oriented Programming Environment*:
  Persistent memory is being used to structure object-oriented programming environments. The programming environments provide support for multi-grained objects inside Clouds objects and support for visibility/migration for these language-defined objects within Clouds objects.

5.2. Reliability in Distributed systems

One of the goals of Clouds is to provide a highly reliable computing environment. The issue of reliability has two parts: maintaining consistency of data in spite of failures; and assuring forward progress for computations. It is necessary to deal with the consistency problem because when a thread executes at several nodes (or several nodes supply objects to a thread because of the DSM abstraction), the results of a computation may be reflected at some nodes but not at others in the event of failure of nodes or communication links. A consistency mechanism should provide the atomicity property which guarantees that a thread computation either completes at all nodes or it has no effect on the state of the system. Thus, if failures are encountered, the effects of all partially completed computation are undone. Consistency by itself does not promise progress because a failure leads to the undoing of the partially completed work. To ensure forward progress, objects and computation must be replicated at nodes with independent failure modes. In the following subsections we briefly touch upon the aspects of the Clouds system that address the consistency and progress requirements.

5.2.1. Notions of Atomicity

The Clouds *"consistency-preservation"* mechanisms present one uniform object-thread abstraction that allows programmers to specify a wide range of atomicity semantics. This

scheme performs automatic locking and recovery of persistent data. Locking and recovery are performed at the segment-level* and not at the object level. Since segments are user defined, this allows user control of the granularity of locking. Custom recovery and synchronization are still possible but will not be necessary in many cases.

Instead of mandating customization of synchronization and recovery for applications that do not need strict atomicity, the new scheme supports a variety of *consistency preserving* mechanisms. The threads that execute are are of two kinds, namely *s-threads* (or *standard* threads) and *cp-threads* (or consistency-preserving threads). The s-threads are not provided with any system-level locking or recovery. The cp-threads on the other hand are supported by well-defined locking and recovery features.

When a cp-thread executes, all segments it reads are read-locked, and the segments it updates are write-locked. Locking is handled by the system, automatically at runtime. The updated segments are written using a 2-phase commit mechanism when the cp-thread completes. Since s-threads do not automatically acquire locks, nor are they blocked by any system acquired locks, they can freely interleave with other s-threads and cp-threads.

There are two varieties of cp-threads, namely the *gcp-thread* and the *lcp-thread*. The gcp-thread semantics provide global (heavyweight) consistency and the lcp-thread semantics provide local (lightweight) consistency. All threads are s-threads when created. Each operation has a static label that declares the consistency needs of the operation. The labels are S (for standard) LCP (for local consistency preserving) and GCP (for global consistency preserving). Various combinations of different consistency labels in the same object (or in the same thread) lead to many interesting (as well as dangerous) execution time possibilities, especially when s-threads update data being read/updated by gcp or lcp threads. The complete discussion of the semantics, behavior and implementation of this scheme is beyond the scope of this paper, and the reader is referred to [Ch*89].

5.2.2. Fault Tolerance

Transaction processing systems provide guarantees about the consistency of data if computations do not complete (due to failures). However, they do not guarantee success of computations. The following section discusses an approach that allows fault-tolerant or resilient computations in Clouds.

The approach uses a mechanism called *parallel execution threads* or PET which tries to provide uninterrupted processing in the face of pre-existing (static) failures, as well as system and software failures that occur while a resilient computation is in progress (dynamic failures) [Ah*88].

To obtain these properties, the basic requirements of the system are:

- Replication of objects, for tolerating static and dynamic failures.

---

* An object may contain multiple data segments. The layout and number of segments are under the control of the user programmer. The segments may contain inter-segment pointers, and objects support dynamic memory allocation on each segment.

- Replication of computation, for tolerating dynamic failures.
- An atomic commit mechanism to ensure correctness.

The PET system works by first replicating all critical objects at different nodes in the system. The degree of replication is dependent on the degree of resilience required.

When a resilient computation is initiated, separate replicated threads (gcp-threads) are created on a number of nodes. The number of nodes is another parameter provided by the user, and reflects the degree of resilience required. The separate threads (or Parallel Execution Threads) run independently as if there is no replication. An invocation by one thread on a replicated object is done by choosing *one* replica of the object and invoking that replica (Figure 5). The replica selection algorithm tries to ensure that separate threads execute at different nodes to minimize the number of threads affected by a failure. After one or more threads complete successfully by executing at operational nodes, one thread is chosen to be the terminating thread. All updates made by this thread are propagated to a quorum of replicas, if available. If there is a failure in committing this thread, another completed thread is chosen. If the commit process succeeds, all the remaining threads are aborted.

This method allows a tradeoff in the amount of resources used (i.e. the number of parallel threads started for each computation) and the desired degree of resilience (number of failures the computation can tolerate, while the computation is in progress.)

6. Concluding Remarks

The goal of Clouds has been to build a general purpose distributed computing environment, suitable for a wide variety of users in the computer science community. We currently have developed a native operating system and an application development environment that is being used for a variety of distributed applications.

Providing a conduit between Clouds and Unix has provided a significant impetus to our development effort. We have saved considerable effort in not having to port program development and environment tools (such as editors and window systems) to a new operating system. Application development can be done in the familiar Unix environment to harness the data and computation distribution capabilities of the new system. The Clouds system has been a fruitful exercise in providing an experimental platform for determining the worthiness of the object-thread paradigm.

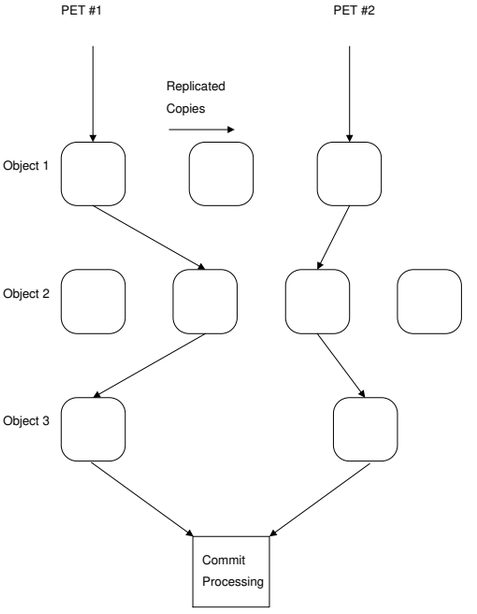

Figure 5: Parallel Execution Threads.

7. Acknowledgements

The authors would like to acknowledge Martin McKendry and Jim Allchin for starting the project and designing the earliest version of Clouds; David Pitts, Gene Spafford and Tom Wilkes for the design and implementation of the kernel and programming support for Clouds, version 1; Jose Bernabeu, Yousef Khalidi and Phil Hutto for their efforts in making the version 1 kernel usable and for the design and implementation of Ra; Sathis Menon R. Ananthanarayanan, Ray Chen and Chris Wilkenloh for significant contribution to the implementation of Clouds version 2, as well as managing the software development effort. Thanks are also due to M. Chelliah, Vibby G., L. Gunaseelan, Ranjit John, Ajay Mohindra, Mark Pearson, Gautam Shah, for their recent participation in and contributions to the project.

8. References

[Ac*86] M. Accetta, R. Baron, W. Bolosky, D Golub, R. Rashid, A. Tevanian and M. Young. *Mach: A New Kernel Foundation for Unix Development*, Proc. Summer Usenix Conference, Usenix, 1986.

[Ah*88] M. Ahamad, P. Dasgupta and R. J. LeBlanc, *Fault-tolerant Atomic Computations in an Object-based Distributed System,* Distributed Computing, Vol 4, no 2, May 1990.

[Al*85] G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe, *The Eden System: A Technical Review*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 1, January 1985, pp 43-58.

[Ch86] D. R. Cheriton, *VMTP: A Transport Protocol for the Next Generation of Communication Systems.* Proceedings of SIGCOMM, 1986.

[Ch88] D. R. Cheriton *The V Distributed System.* Communications of the ACM, March 1988.

[Ch*89] R. Chen and P. Dasgupta, *Linking Consistency with Object/Thread Semantics: An Approach to Robust Computations*, 9th International Conference on Distributed Computing Systems, Newport Beach CA, June 5th-7th, 1989.

[Da*90a] P. Dasgupta et. al. *The Design and Implementation of the Clouds Distributed Operating System.* Usenix Computing Systems Journal, Volume 3, Number 1, 1990.

[Da*90b] P. Dasgupta and R. C. Chen, *Memory Semantics in Persistent Object Systems*, chapter in Implementation of Persistent Object Systems, Editor: Stan Zdonick, Morgan Kaufman Publishers, 1990.

[Lis87] Liskov, B. *Distributed Programming in ARGUS.* CACM, March, 1988.

[Li*89] K. Li and P. Hudak, *Memory Coherence in Shared Virtual Memory Systems* ACM Transactions on Computer Systems, Vol. 7, No 4, November 1989.

[Mu*90] S. J. Mullender, G. Rossum, A. S. Tannenbaum, R. Renesse and H. Staveren. *Amoeba: A Distributed Operating System for the 1990s*, Computer, IEEE, May 1990.

[Wu*74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson and F. Pollack. W. A. Wulf et. al. *HYDRA: The Kernel of a Multiprocessor Operating System*, Communications of the ACM, (17,6) June 1974.