The Clouds Distributed Operating System

Partha Dasgupta, Arizona State University

Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran, Georgia Institute of Technology

Clouds is a general-purpose operating system for distributed environments. It is based on an object-thread model adapted from object-oriented programming.

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 it provides, a distributed operating system is classified as general purpose, real time, or embedded.

The need for distributed operating systems stems from rapid changes in the hardware environment in many organizations. Hardware prices have fallen rapidly in the last decade, resulting in the proliferation of workstations, personal computers, data and compute servers, and networks. This proliferation has underlined the need for efficient and transparent management of these physically distributed resources (see sidebar titled "Distributed operating systems").

This article presents a paradigm for structuring distributed operating systems, the potential and implications this paradigm has for users, and research directions for the future.

Two paradigms

Operating system structures for a distributed environment follow one of two paradigms: message-based or object-based. *Message-based* operating systems place a message-passing kernel on each node and use explicit messages to support interprocesses communication. The kernel supports both *local communication* (between processes on the same node) and *nonlocal* or *remote communication*, which is sometimes implemented through a separate 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, the requests are via message passing. Message-based operating systems are attractive for structuring operating systems because the policy, which is encoded in the server processes, is separate from the mechanism implemented in the kernel.

Object-based distributed operating systems encapsulate services and resources into entities called objects. *Objects* are similar to instances of abstract data types. They are written as individual modules composed of the specific operations that define the module interfaces. Clients request access to system services by invoking the appropriate system object. The invocation mechanism is similar to a protected procedure call. Objects encapsulate functionality much as server processes do in message-based systems (see sidebar titled "What can objects do?").

Among the well-known message-based systems are the V-system [1] developed at Stanford University and the Amoeba system[2] developed at Vrije University in Amsterdam, Netherlands. These systems provide computation and data services via servers that run on machines linked by a network.

Most object-based systems are built on top of an existing operating system, typically Unix. Examples of such systems include Argus,[3] Cronus,[4] and Eden.[5] These systems support objects that respond to invocations sent via the message-passing mechanisms of Unix.

Mach[6] is an operating system with distinctive characteristics. A Unix-compatible system built to be machine-independent, it runs on a large variety of uniprocessors and multiprocessors. It has a small kernel that handles the virtual memory and process scheduling, and it builds other services on top of the kernel. Mach implements mechanisms that provide distribution, especially through a facility called memory objects, for sharing memory between separate tasks executing on possibly different machines.

Distributed operating systems

Networked computing environments are now commonplace. Because powerful desktop systems have become affordable, 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 that are not present in centralized systems. To keep user productivity high, the distribution must be transparent and the environment must appear centralized. The short-term solution adopted by current commercial software extends conventional operating systems to allow transparent file access and sharing. For example, Sun added Network File System 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 — for example, Microsoft Windows 3.0, Novell Netware, Appletalk, and PC-NFS.

A better long-term solution is the design of an operating system that considers the distributed nature of the hardware architecture at all levels. A distributed operating system is such a system. It makes a collection of distributed 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.

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 that functions as a repository for long-lived (that is, persistent) data. A *user workstation* is a machine that provides a programming environment for developing applications and an interface with the compute and data servers for executing those applications on the servers. Note that when a disk is associated with a compute server, it can also act as a data server for other compute servers. Clouds is a native operating system that runs on top of a native kernel called Ra (after the Egyptian sun god). 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 ap-

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 routines to complex algorithms, from shared library accesses to elaborate system services.

Objects can also provide specialized services. For example, an object can represent a sensing device, and an invocation can gather data from the device without knowing about the mechanisms involved in accessing it — or its location. Similarly, an object that handles terminal or file I/O contains defined read and write operations.

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 can monitor an object's environment and can inform some other entity (another object) when an event occurs. This feature is particularly useful in objects that manage sensor-monitoring devices.

Objects are a simple concept with a major impact. They can be used for almost every need — from general-purpose programming to specialized applications — yet provide a simple procedural interface to the rest of the system.

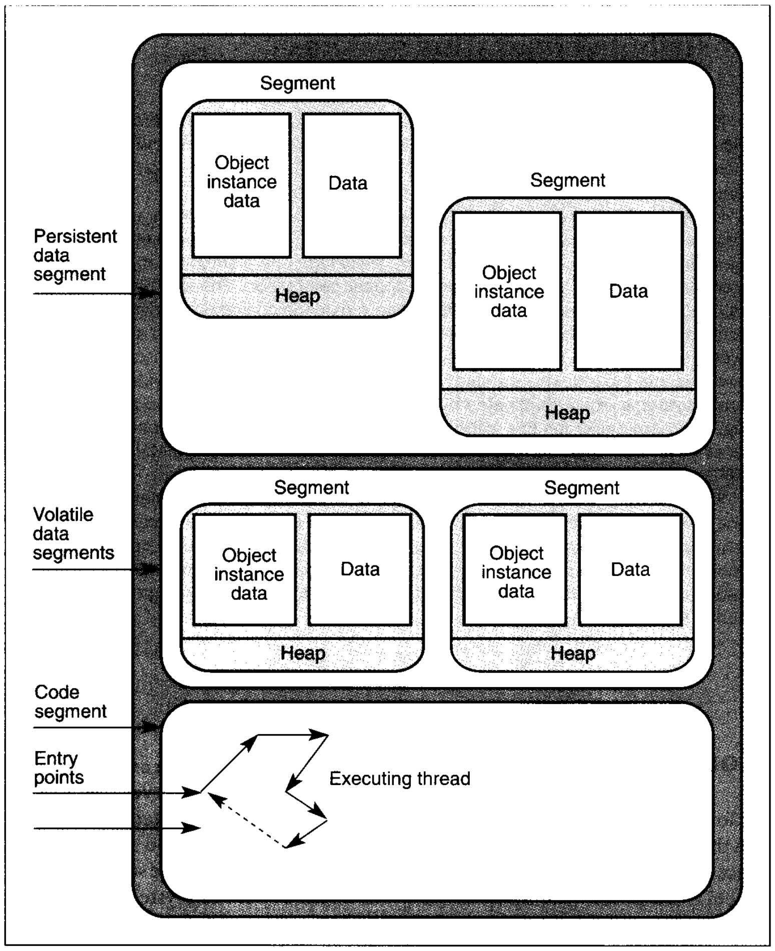


Figure 1. A Clouds object.

plications, 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 system resources.

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*. Each object is an instance of an abstract data type consisting of data and operations on the data. The operations are called *methods*. The object type is defined by a *class*. A class can have zero 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, which in turn accesses or updates data stored in the object and may trigger sending messages 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 results from 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, interprocess communication, information sharing, and long-term storage. This model has been further augmented to support atomicity and reliable execution of computations.

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

Clouds paradigm

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

Objects. A Clouds object is a persistent (or nonvolatile) 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 unless explicitly deleted — like a file. As the following description of objects shows, Clouds objects are somewhat "heavyweight," that is, 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 address 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 object's persistent data structures (see Figure 1). Recall that the data in the object can be manipulated only from within the object. Data can pass into the object when an entry point is invoked (input parameters). Data can pass 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 — a naming service translates them to sysnames. Objects are physically stored in data servers but are accessible from all compute servers in the system, thus providing location transparency to the users.

Threads. The only form of user activity in the Clouds system is the user thread. A thread is a logical path of execution that traverses objects and executes the code in them. 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 and results are strictly data; they cannot be addresses. This restriction is mandatory because addresses in one object are meaningless in the context of another object. In addition, object invocations can be nested as well as recursive. After the thread completes execution of the operation it was created to execute, it terminates.

The nature of Clouds objects prohibits a thread from accessing 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 executions in the Clouds object spaces. The programmer uses system-supported primitives such as locks or semaphores to handle concurrency control within the object.

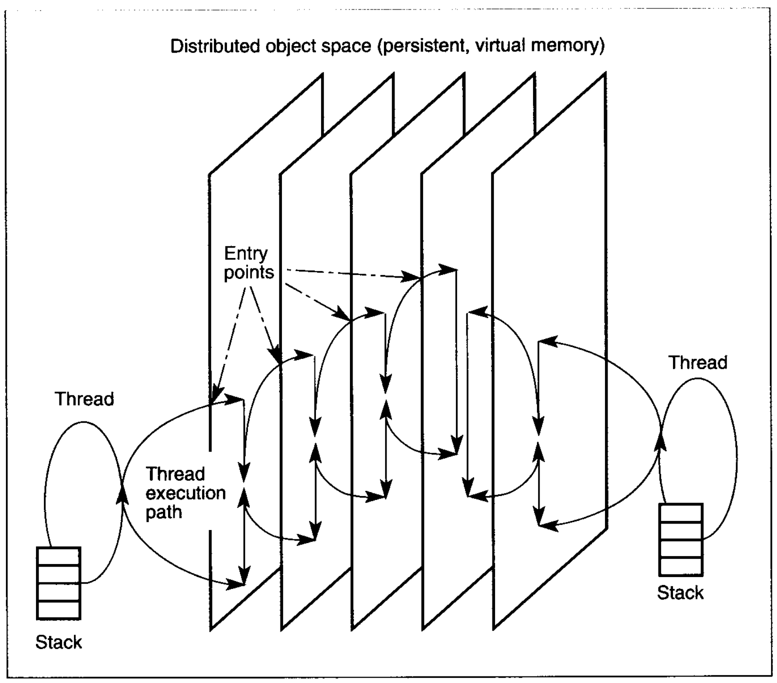


Figure 2. Distributed object memory.

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 and is volatile (that is, 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 a persistent address space. This unification makes programming simpler. Persistent objects provide a structured single-level store that is cosmetically similar to mapped files in Multics and SunOS.

Some systems use message-passing for communicating shared data and coordinating computations. Clouds shares data by placing the data in an object. Computations that need access to shared data invoke the object where the data exists. Clouds does not support messages and files at the operating system level, although it does allow objects to simulate them if necessary (see sidebar titled "No files? No messages?"on the next page).

In a message-based system, the user must determine the desired level of concurrency at the time of writing an application, and program it as a certain number of server processes. The object-thread model of Clouds eliminates this requirement. An object can be written from the viewpoint of the functionality it is meant to provide, rather than the actual level of concurrency it may have to support. At execution time, the level of concurrency is specified by creating concurrent threads to execute in the objects that compose the user-level application. The application objects, however, must 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.

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. An object can store data in any form 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 those used for storage, and
- Allow controlled concurrent access to shared data, without regard to data location.

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

In fact, objects that store byte-sequential data can simulate files, and they can 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 functional equivalence of messages and shared memory is well known. If desired, a buffer object with defined send and receive invocations can serve as a port structure between two (or more) communicating processes.

We feel that files, messages, and disk I/O are artifacts of hardware structure. Given an object implementation, these features are neither necessary nor attractive. New programming paradigms based on object-oriented styles use persistent memory effectively. They do not use files and messages.

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.

Programming in the Clouds model. For the programmer, Clouds has two kinds of objects: classes and instances. A *class* is a template used to generate instances. An *instance* is an object invocable by user threads. Thus, to write application programs for Clouds, a programmer writes one or more Clouds classes that define the application code and data. The programmer can 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.

The following is a simple example of programming in the Clouds system. 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 definition is shown in Figure 3a.

Once the class is compiled, any number of instances can 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. A command in the Clouds command line interpreter can call the entry point in the object. Entry points can 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). A name server then translates the user name to a sysname. (Recall that a sysname is a unique name for an object, which is needed for invoking the object.) The code fragment in Figure 3b details the steps in accessing the Clouds object Rect01 and invoking operations on it.

Clouds provides a variety of mechanisms to programmers. These include 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.

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

User objects and their entry points are typed by the language definition. The compiler performs static type-checking on the object and entry point types at compile time. It performs no runtime type checking. Clouds objects are coarse-

```
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
                                                                  (a)

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
                                                                  (b)
```

Figure 3. Example code for Clouds.

grained, unlike the fine-grained entities found in such object-oriented programming languages as Smalltalk. Since an object invocation in Clouds is at least an order of magnitude more expensive than a simple procedure call, it is appropriate to use a Clouds object 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. DC++ is an extension of C++ that systems programmers use. Distributed Eiffel is an extension of Eiffel that targets application developers. The design of both DC++ and Distributed Eiffel supports persistent fine-grained and large-grained objects, invocations, thread creation, synchronization, and user-level object naming.

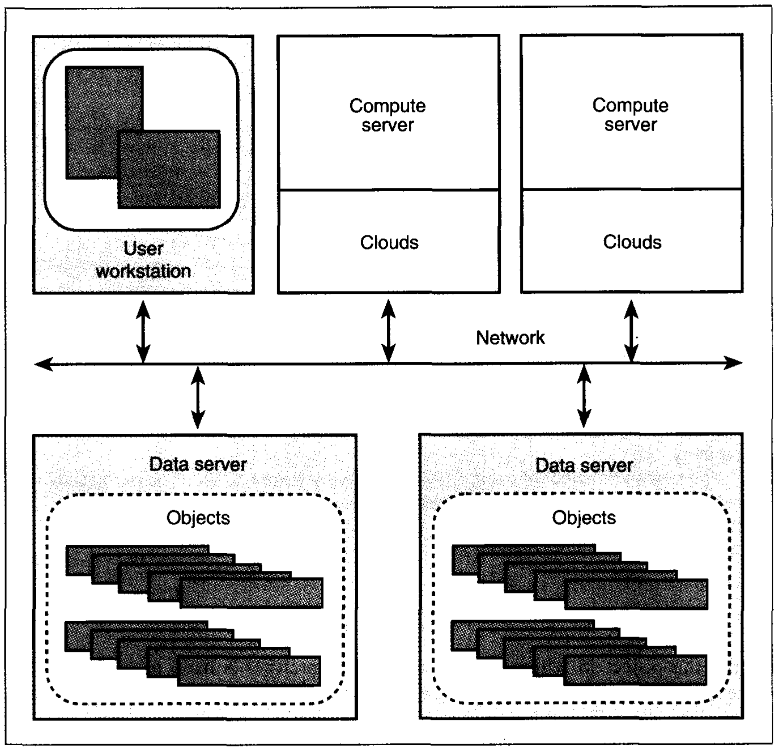


Figure 4. Clouds system architecture.

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 configured as three logical categories of machines, each supporting a different logical function. These are compute servers, data servers, and user workstations.

The system's core 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. Data servers provide secondary storage. They store Clouds objects and supply the objects' code and data to the compute servers. The data servers also support the distributed synchronization functions. The third machine category, the user workstation, provides user access to Clouds compute servers. Compute servers, in turn, know how and when to access data servers.

The logical machine categories do not require a one-to-one scheme for mapping to physical machines. Although a diskless machine can function only as a compute server, a machine with a disk can simultaneously act as a compute and data server. This enhances computing performance, since data accessed via local disk is faster than data accessed over a network. However, in our prototype system, shown in Figure 4, we use a one-to-one mapping to simplify the system's implementation and configuration.

A suite of programs that run on top of Unix on Sun workstations provide the user interface to Clouds. These programs include Distributed Eiffel and DC++ compilers, a Clouds user shell (under X Windows), a user I/O manager, and various utilities. The user interface with these programs is through the familiar Unix utilities (including Unix editors).

User environment. A user writes Clouds programs in DC++ 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 node (or user on a Unix machine) can create instances of these classes and generate invocations to the objects thus created. Note that once created, the objects become part of the persistent object memory and can be invoked until they are explicitly deleted.

A user invokes an Clouds object by specifying the object, the entry point, and the arguments to the Clouds shell. The 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.

System environment. As we mentioned earlier, the hardware environments 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, the load at each compute server, and the availability of resources needed for the computation. Once this decision has been made, the second task is to bring into the selected compute

Distributed shared memory

The name space represented by the Clouds objects constitutes a shared address space. Since each object consists of a linear address space, these two spaces in conjunction provide a systemwide two-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 occurs through a mechanism that we call distributed shared memory (DSM). When a thread on node *A* invokes an operation on object *O*, the invocation executes on node *A*. If *O* is not located on *A*, a series of page faults occurs, 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*, both *A* and *B* must always see the exact same contents of *O*. This is called one-copy semantics; it is maintained by coherence protocols that are an integral part of the DSM access strategy.

The perceived effect of using DSM 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.

server the object in which the thread executes. This requires a remote-paging facility. 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 must also maintain the consistency of shared pages. This is satisfied in Clouds by a mechanism called *distributed shared memory*. DSM supports the notion of shared memory on a nonshared memory (distributed) architecture (see sidebar titled "Distributed shared memory"). The data servers execute a coherence protocol that preserves single-copy semantics for all the objects.[7] 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 a 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 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 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 found in other systems such as the V system,[1] but it 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 consistently by concurrent threads even if they are executing on different compute servers. The distributed synchronization support provided by data servers allows threads to synchronize their actions regardless of where they execute.

Clouds implementation

The Clouds implementation uses a *minimalist* approach towards operating system development (like the V system, Amoeba, and Mach 3.0). With this approach, each level of the implementation consists of only those functions that cannot be implemented at a higher level without a significant performance penalty. Traditional systems such as Unix provide most operating-system services in one big monolithic kernel. Unlike such systems, we differentiate between the operating-system kernel 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 a high level using the lower level mechanisms.

The current Clouds implementation has three levels. At the lowest level is the minimal kernel, Ra, which provides the mechanisms for managing 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 noncritical services such as naming and spooling are implemented as user objects to complete the functionality of Clouds.

The Ra kernel. This native minimal kernel supports virtual memory management and low-level scheduling. Ra implements four abstractions:

- *Segments*. A segment is a variable-length sequence of uninterpreted bytes that exists either on the disk or in physical memory. Segments have systemwide unique names and, once created, segments persist until they are 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. A segment can be mapped to a contiguous range of addresses in a virtual space.
- *IsiBas*. An IsiBa is the abstraction of system activity and can be thought of as a lightweight process. (Its name comes from ancient Egyptian: Isi means light, and Ba means soul.) An IsiBa is simply a kernel resource that is associated with a stack to realize a schedulable entity. There are several types of stacks in the system (for example, kernel, interrupt,

and user), and an IsiBa can use an instance of any stack type. 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 nonvolatile data storage for segments. A Clouds compute server has access to one or more partitions, but a segment belongs to exactly one partition. 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.

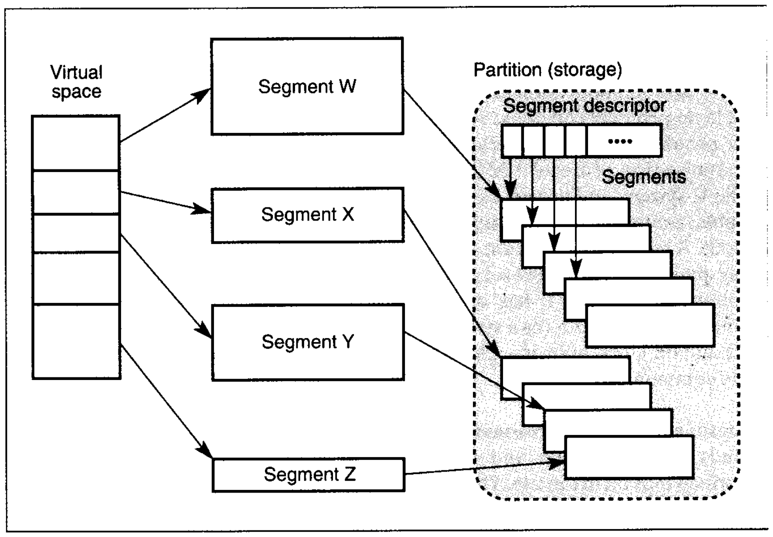


Figure 5. Virtual spaces, segments, and partitions.

Figure 5 shows the relationship between virtual spaces, segments, and partitions.

The implementation of Ra is separated into machine-dependent and machine-independent parts. All Ra components use the class mechanisms of C++, a scheme that enhances Ra's object structure. Ra consists of 6,000 lines of machine-dependent C++ code, 6,000 lines of machine-independent C++ code, and 1,000 lines of Sun (68020) assembly code. It currently runs on the Sun-3 class machines. More details about the implementation are in Dasgupta et al.[8]

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 Ra-defined operations. Ra exports these operations as kernel classes that are inherited by the system objects. Conceptually, the system objects are similar to Clouds objects that live in their own virtual space. 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 the 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; these functions include the buffer manager, uniform I/O interface, and 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, object manager, and user I/O manager.

The following bulleted paragraphs describe some of the important system objects.

• *Thread manager.* As mentioned earlier, a thread can 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 I/O requests are made during the computation. The thread manager is responsible for the creation, termination, naming, and 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 invoking thread 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 used by the Clouds nodes. When node *A* needs a page of data, 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 maintenance of 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 written data 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.* Two system objects handle networking: the Ethernet driver and the network protocol. All Clouds communication uses a transport layer protocol called the Ra transport protocol. RaTP is similar to the communication protocol VMTP[9] 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.

Status and current performance. The Clouds implementation and features described thus far are in use. The kernel performance is good. Context-switch time is 0.15 milliseconds. The time to service a page fault when the page is resident on the same node costs 2.3 ms for a zero-filled, 8-kilobyte page; it costs 1.5 ms for a nonzero-filled page.

Networking is one of Clouds' most heavily used subsystems, 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 1.59 ms; this involves sending and receiving a short message (72 bytes) between two compute servers. The RaTP reliable round-trip time is 3.56 ms. To transfer an 8-kilobyte page reliably from one machine to another costs 12.3 ms, compared to 70 ms using Unix file-transfer protocol and 50 ms using Sun-NFS.

Object invocation costs vary widely depending upon whether the object is currently in memory or has 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 cost is much closer to the minimum than the maximum.

Clouds and distributed systems research

The Clouds project includes continuing systems research on several topics.

Using persistent objects. Persistent, shared, single-level storage is the central theme of the Clouds model. Therefore, the thrust of several related research projects was to effectively support and exploit persistent memory in a distributed setting. Another area of research is in harnessing the distributed resources to speed up the execution of specific applications compared to a single-processor implementation. We summarize some of these projects here.

Distributed programming. Using the DSM feature of Clouds, centralized algorithms can 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 powerful programming support in the Clouds system.[10]

Lisp programming environment. If the address space containing a Lisp environment can be made persistent, several advantages accrue, including not having to save/load the environment on startup and shutdown. Further, invoking entry points in remote Lisp interpreters allows interenvironment operations that are useful in building knowledge bases. Other features that naturally arise from the distributed nature of the system include concurrent evaluations and load sharing. An implementation of the Clouds Lisp Distributed Environment (Clide) is currently in experimental usage.

Object-oriented programming environment. Persistent memory is being used to structure object-oriented programming environments. These environments support multigrained objects inside Clouds objects and visibility/migration for these language-defined objects within Clouds objects.

Reliability in distributed systems. One goal 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. A consistency problem can occur when a thread executes at several nodes or several nodes supply objects to a thread because of the DSM abstraction. In that case, a node or communication link failure causes the computation results to be reflected at some nodes but not at others. A consistency mechanism should provide the atomicity property guaranteeing that a thread computation either completes at all nodes or has no effect on system state. Thus, if failures occur, the effects of all partially completed computations are undone.

Consistency by itself does not promise progress, because a failure undoes the partially completed work. To ensure forward progress, objects and computation must be replicated at nodes with independent failure modes. The following aspects of the Clouds system address the consistency and progress requirements.

Atomicity. The Clouds *consistency-preservation* mechanisms present one uniform object-thread abstraction that lets programmers specify a wide range of atomicity semantics. This scheme, called Invocation-Based Consistency Control, automatically locks and recovers persistent data. Locking and recovery are performed at the segment level, not at the object level. (An object can contain multiple data segments. The layout and number of segments are under the control of the user programmer. The segments may contain intersegment pointers, and objects support dynamic memory allocation on each segment.) Because segments are user-defined, the user can control the granularity of locking. Custom recovery and synchronization are still possible but are unnecessary 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 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. The system automatically handles locking at runtime. The updated segments are written by a two-phase commit mechanism when the cp-thread completes. Because 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 its consistency needs. 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. (For a discussion of the semantics, behavior, and implementation of this scheme, see Chen and Dasgupta.[11])

Fault tolerance. Transaction-processing systems guarantee data consistency if computations do not complete (due to failures). However, they do not guarantee computational success. The Clouds approach to fault-tolerant or resilient computations is called *parallel execution threads*. PET 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).[12]

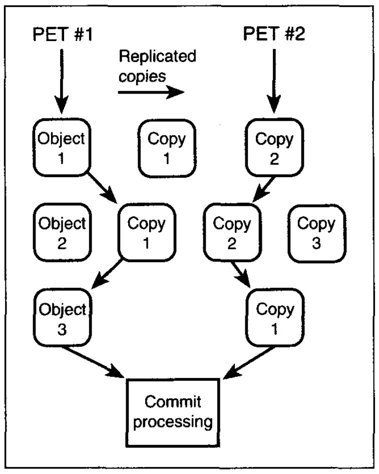


Figure 6. Parallel execution threads.

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

- replication of objects, for tolerating static and dynamic failures;
- replication of computation, for tolerating dynamic failures; and
- 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 depends on the degree of resilience required.

Initiating a resilient computation creates separate replicated threads (gcp-threads) 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 PETs) run independently as if there were no replication. A thread invokes a replicated object by choosing certain copies of the object, as shown in Figure 6. 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 trade-off in the amount of resources used (that is, number of parallel threads started for each computation) and the desired degree of resilience (that is, number of failures the computation can tolerate, while the computation is in progress).

The goal of Clouds was to build a general-purpose, distributed, computing environment suitable for a wide variety of users in the computer science community. We 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 saved us considerable effort. We did not have to port program development and environment tools (such as editors and window systems) to a new operating system, and we can develop applications that harness the new system's data and computation distribution capabilities in the familiar Unix environment. The Clouds system has been a fruitful exercise in providing an experimental platform for determining the worthiness of the object-thread paradigm. ■

References

1. D.R. Cheriton, "The V Distributed System," *Comm. ACM*, Vol. 31, No. 3, Mar. 1988, pp. 314-333.
2. S.J. Mullender et al., "Amoeba: A Distributed Operating System for the 1990s," *Computer*, Vol. 23, No. 5, May 1990, pp. 44-53.
3. B. Liskov, "Distributed Programming in Argus," *Comm. ACM*, Vol. 31, No. 3, Mar., 1988, pp. 300-313.
4. R.E. Schantz, R.M. Thomas, and G. Bono, "The Architecture of the Cronus Distributed Operating System," *Proc. Sixth Int'l Conf. on Distributed Computing Systems*, CS Press, Los Alamitos, Calif., Order No. 697, 1986, pp. 250-259.
5. G.T. Almes et al., "The Eden System: A Technical Review," *IEEE Trans. Software Eng.*, Piscataway, N.J., Vol. SE-11, No. 1, Jan. 1985, pp 43-58.

6. M. Accetta et al., "Mach: A New Kernel Foundation for Unix Development," *Proc. Summer Usenix Conf.*, Usenix, 1986, 93-112.

7. K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems*, Vol. 7, No. 4, Nov. 1989, pp. 321-359.

8. P. Dasgupta et al., "The Design and Implementation of the Clouds Distributed Operating System," *Usenix Computing Systems J.*, Vol. 3, No. 1, Winter 1990, pp. 11-46.

9. D.R. Cheriton, "VMTP: A Transport Protocol for the Next Generation of Communication Systems." *Proc. SIGcomm*, 1986, pp. 406-415.

10. P.Dasgupta and R.C. Chen, "Memory Semantics in Persistent Object Systems," in *Implementation of Persistent Object Systems*, Stan Zdonick, ed., Morgan Kaufmann Publishers, San Mateo, Calif., 1990.

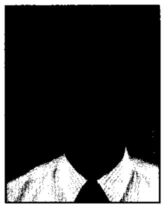
11. R. Chen and P. Dasgupta, "Linking Consistency with Object/Thread Semantics: An Approach to Robust Computations," *Proc. Ninth Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., 1989, Order No. 1953, pp. 121-128.

12. 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, pp. 69-80.

Acknowledgments

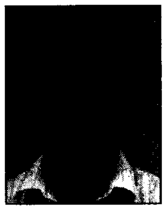
This research was partially supported by the National Aeronautics and Space Administration under contract NAG-1-430 and by the National Science Foundation grants DCS-8316590 and CCR-8619886 (CER/II program).

The authors 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 for managing the software development effort. Ray Chen and Mark Pearson designed and implemented IBCC and Clide, respectively. We also thank M. Chelliah, Vibby Gottemukkala, L. Gunaseelan, Ranjit John, Ajay Mohindra, and Gautam Shah for their participation in and contributions to the project.



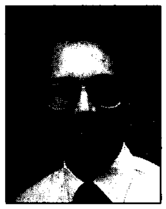
Partha Dasgupta is an associate professor at Arizona State University. He was technical director of the Clouds project at Georgia Institute of Technology, as well as coprincipal investigator of the institute's NSF-Coordinated Experimental Research award. His research interests include distributed operating systems, persistent object systems, operating system construction techniques, distributed algorithms, fault tolerance, and distributed programming support.

Dasgupta received his PhD in computer science from the State University of New York at Stony Brook. He is a member of the IEEE Computer Society and ACM.



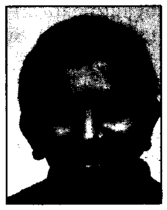
Mustaque Ahamad is an associate professor in the School of Information and Computer Science at the Georgia Institute of Technology, Atlanta. His research interests include distributed operating systems, distributed algorithms, fault-tolerant systems, and performance evaluation.

Ahamad received his BE with honors in electrical engineering from the Birla Institute of Technology and Science, Pilani, India. He obtained an MS and a PhD in computer science from the State University of New York at Stony Brook in 1983 and 1985.



Richard J. LeBlanc, Jr. is a professor in the School of Information and Computer Science at the Georgia Institute of Technology. As director of the Clouds project, he is studying language concepts and software engineering methodology for a highly reliable, object-based distributed system. His research interests include programming language design and implementation, programming environments, and software engineering.

LeBlanc received his BS in physics from Louisiana State University in 1972 and his MS and PhD in computer sciences from the University of Wisconsin-Madison in 1974 and 1977. He is a member of the ACM, the IEEE Computer Society, and Sigma Xi.



Umakishore Ramachandran is an associate professor in the College of Computing at the Georgia Institute of Technology. He is currently involved in several research projects, including hardware/software trade-off in the design of distributed operating systems. He participated in the design of several distributed systems, including Charlotte at University of Wisconsin-Madison, Jasmin at Bell-Core, Quicksilver at IBM Almaden Research Center, and Clouds. His primary research interests are in computer architecture and distributed operating systems.

Ramachandran received his PhD in computer science from the University of Wisconsin-Madison in 1986. In 1990, he received an NSF Presidential Young Investigator Award. He is a member of the ACM and the IEEE Computer Society.

Readers may contact Partha Dasgupta at the Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287; e-mail partha@enuxha.eas.asu.edu.