Originally published at pp. 597-603 of Coulouris, Dolllimore & Kindberg, Distributed Systems, Edition 2, 1994.

Clouds

History and architectural overview

Clouds [Dasgupta *et al*. 1991] is an operating system for the support of distributed objects, developed at the Georgia Institute of Technology, US. The first version of Clouds was implemented in 1986. We describe version 2, which has been developed since 1987. This version is based upon a microkernel, called *Ra*, that has been designed to support a variety of distributed object programming models. Clouds incorporates a complete set of system-level facilities, including storage and input-output. It is based on an object-thread paradigm, which is a distributed operating system version of the object-oriented programming paradigm. Objects are heavyweight, passive entities, each with its own address space that encapsulates code and data. Threads are activities that execute the methods within objects, invoking operations on further objects as they do so.

Each Clouds object is represented by one persistent replica held at a server computer, and possibly several volatile replicas at computers where operations upon the object are being executed. Distributed shared memory techniques are employed so that the memory inside a single object being shared at different computers is rendered consistent (see Chapter 17).

In the Mach, Chorus and Amoeba distributed operating systems, shared resources are managed by servers. These systems are object-based in the sense that clients can refer to all resources in a uniform manner.

Clouds distinguishes itself from Amoeba, Mach and Chorus in its object-oriented style of invocations and the type of threads it supports.

As we shall see, Clouds does not provide message passing; and although objects are akin to the execution environments found in process-based operating systems, objects are not associated with their own threads. Threads in Clouds are activities that can move from object to object and thus from computer to computer.

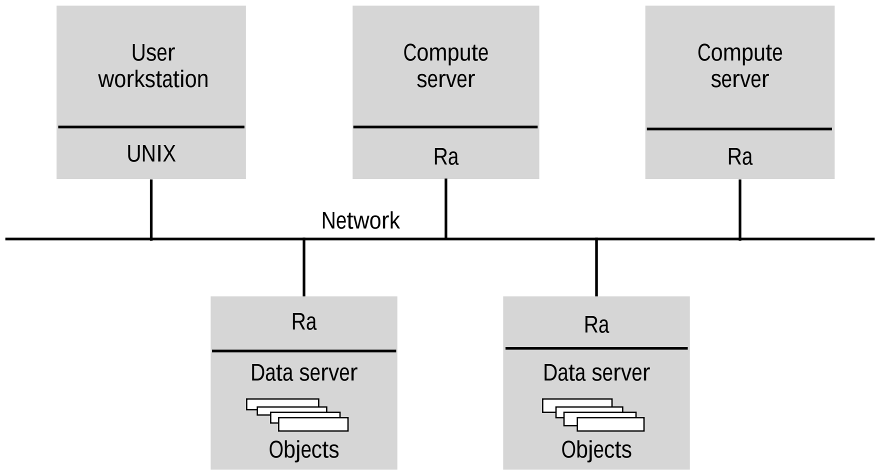
Clouds is designed to support a range of languages based on the object-thread paradigm. As currently implemented, it incorporates language support for an extension of C++ [Stroustrup 1986] called DC++, and an extension of Eiffel [Meyer 1988] called Distributed Eiffel. Programmers can create and compile classes using these languages, and create *instances* – that is, objects with private state – from these classes. Interactive users can create threads implicitly by specifying object invocations from the command line. Clouds executes threads and carries out invocations in a network-transparent fashion.

The Clouds system architecture (Figure 1) consists of three classes of computer: workstations, data servers and compute servers.

Workstations are used for interaction by the users, and these run the UNIX operating system. UNIX provides a file service and user interaction service to the rest of the Clouds implementation.

Data servers manage secondary storage for the code and data belonging to objects. Data servers run the Ra kernel. No file service is presented to programmers in Clouds, which has no special file concept. Instead, all objects in Clouds are themselves persistent. Objects, like

Figure 1 The Clouds system architecture.



files, outlive any threads that perform invocations upon them, until they are explicitly deleted. We shall explore this notion further, below.

Compute servers are, taken together, functionally similar to the processor pool in Amoeba, although they need not be rack-based and can be entire computers. Clouds threads execute operations on objects at compute servers that are dynamically chosen on an invocation-by-invocation basis. The compute server computers are homogeneous, and do not need any secondary store. They run the Ra kernel.

The data server and compute server roles are logical. In principle, a single computer with secondary storage attached could play both roles.

Design goals and chief design features

Clouds has the following major design goals and features:

Support for object-thread computational model: Clouds objects are abstractions of protected, passive storage, and threads are activity abstractions which exist independently of objects. Since Clouds objects are heavyweight (they include a virtual address space), it is envisaged that language run-time support software can implement many fine-grained objects (of size in the order of tens or hundreds of bytes) inside a single Clouds object. These small objects have language-specific semantics, which may differ from the basic Clouds object model. The clustering of objects inside host Clouds objects reduces the average object creation and deletion costs. Since some invocations in general will be between objects within the same Clouds object, clustering can also reduce the number of context switches consequent upon invocations.

Network-transparent object invocation: Direct access to the code or data within an object is prevented by memory management hardware. The only mechanism to access the state of an object is that of invocation. An invocation specifies a target object, the method to be called within it, and the input and output parameters. A thread making an invocation is blocked until the corresponding method has been executed and any result parameters sent back. All data transfer between objects occurs, from the programmer's point of view, by invocation parameter passing, and not by message passing such as that provided by say, Amoeba and Mach. Invocation is network transparent, and a thread can execute an object invocation at any compute server – regardless of where the object is stored.

Persistent, single-level storage: To the programmer, there is only a single level of storage instead of the usual primary/secondary storage hierarchy. Changes made to an object's data are, with certain exceptions, automatically reflected in a version held on secondary store at

Figure 2 Invocations upon a rectangle object.

```
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

rectangle_ref rect;               // ‘rect’ is a variable that holds
                                  // an instance of a class that refers
                                  // to an object of type rectangle

rect.bind("RectA");               // call to name server, which returns
                                  // sysname for existing object "RectA"
rect.size(5,10);                  // invocation of RectA
printf("%d\n", rect.area());      // will print 50
```

a data server. This mechanism is akin to mapped files. Objects do not have to be explicitly saved, or marshalled to any special form (such as a sequence of bytes) in order to be stored. Of course, for an object operation to be executed, Clouds must fetch the object’s code and data into a compute server’s primary memory.

Sharing via objects: All sharing in Clouds takes place by performing invocations upon common objects – just as sharing occurs through files in other systems. Threads executing methods inside the same object share the object’s code and data. Clouds therefore provides low-level concurrency control mechanisms such as semaphores. However, to complicate matters further from the implementation point of view, threads executing at separate computers can execute concurrently within the same object (this is transparent to the programmer). The threads access the same code and data at the same virtual addresses, even though the computers at which they execute do not share physical memory. Memory can indeed be *logically* shared between threads that execute at different computers without *physical* shared memory, through the abstraction of distributed shared memory. Techniques for implementing distributed shared memory are described in Chapter 17.

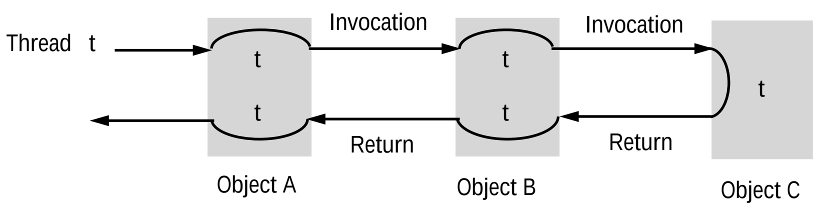
Automatic load balancing: Clouds enables a system-supported load balancing policy to be exercised whenever an invocation is made, to choose the compute server to execute the method on the target object. Alternatively, the programmer may specify locations for invocation execution.

Objects, threads and invocations

An object’s virtual address space consists of regions that are mapped to memory resources called *segments*. An object contains persistent segments for code, data and a heap. In fact, an object typically contains two heaps. The persistent heap is used for dynamically allocated memory that persists with the object. An object can also contain a volatile segment for temporary heap data. The contents of a volatile segment can be discarded by the system as long as no thread executes within the object that contains them. All segments are accessed by direct memory references: it is the responsibility of the virtual memory implementation to read or write the corresponding data from or to secondary storage as appropriate.

Objects are referenced by globally unique, location-independent identifiers called *sysnames*. A thread performing an invocation must supply the target object’s sysname. It obtains the sysnames of shared objects through a name server, which stores a mapping of textual names to sysnames. An invocation specifies a method identifier which is used to look up an *entry point* in the target object – an address from which the system continues the execution of the invoking thread

Figure 3 A single thread performs invocations on three objects.



when it enters the invoked object. Clouds assumes static (compile-time) type checking is used to ensure that only legitimate entry points in legitimate objects are accessed by threads.

Figure 2 (adapted from Dasgupta *et al*. [1991]) shows some example DC++ code for manipulating a rectangle object which is an instance of the class *rectangle*. After looking up the named instance "RectA" from the name server, the code makes two further invocations upon this rectangle, to set its size and determine its area. Note that, corresponding to the *rectangle* class, the language automatically provides the *rectangle_ref* class of references to rectangle instances. This provides the *bind* method, which makes a call to the name server to look up the sysname from a textual name, as well as inheriting the methods of the underlying *rectangle* class.

Figure 3 shows a thread invoking operations upon a succession of objects, and returning after each invocation has been executed, eventually to the original invoking object. Invocation is synchronous. If a thread is required to continue executing in an object while another invocation is performed, it can create a second thread to perform the invocation while the original thread continues.

The movement of a single thread between objects in separate address spaces is known as *thread diffusion*. Of course, this view is an abstraction: thread diffusion is implemented using familiar activity and communication abstractions provided by the underlying Ra kernels. However, what does differ fundamentally from the previous computational models we have considered is that Clouds programmers do not control the number of threads within an object. Whereas, for example, a Mach programmer explicitly creates the threads assigned to receiving incoming RPCs arriving at a task, threads in Clouds enter an object ‘from outside’. Potentially as many threads execute within an object as there are invocations to be executed. Of course, there is a physical resource limit on the number of threads that can execute at the same compute server, imposed by the Ra kernel.

Objects do not contain their own stacks. Instead, each thread is associated with a single stack, and this stack is installed in an object’s address space when the thread performs an invocation upon the object. We now look at the implementation of the invocation mechanism and the single-level store model in more detail.

Implementation of the computational model

The Ra microkernel provides three basic abstractions:

isibas are the activity abstractions, which execute either in user mode or supervisor mode but which are confined, unlike the higher-level threads, to a single computer;

segments are mapped memory resources;

virtual spaces consist of collections of regions mapped to segments.

Virtual spaces are not themselves entire address spaces. Each isiba can be associated with up to three distinct virtual spaces: *O space* contains the object code and data for the current user-level invocation; *P space* contains thread-specific, private data such as a stack and a parameter-passing area; and *K space* contains an image of the kernel itself and is common to all isibas.

In addition to the kernel, K space also contains system objects. These are objects that perform some higher-level service function than the standard kernel. System objects are the servers of the Clouds system. System objects execute in the kernel’s address space for the sake of efficiency. They are separately compiled, and loaded into the kernel at configuration time. They, like the kernel itself, are written in the C++ language. They inherit operations from the kernel's classes, and thus can invoke its services directly (whereas user-level isibas must make system call traps). Unlike Chorus servers, there is no facility for loading system objects dynamically.

System objects are used to implement network communication, and to control input-output between Ra nodes and the users' workstations. The main system objects of relevance here, however, are the *thread manager* and *user object manager*.

Performing an invocation ◊ First, unless the programmer has specified a target node, a compute server is chosen to execute an invocation upon a given object. If the chosen server differs from the one where the thread currently resides, then an RPC is made to the destination, requesting it to perform the invocation. The user object manager at the compute server creates the object's virtual space, *O* space, if it does not already exist there. It is able to do this using the object's virtual space descriptor segment, stored on a data server. It adds a *P* space containing the invoking thread's stack, copied from the invoking computer, and a parameter-passing area. The thread manager, a system object that records information about threads as they execute across the distributed system, is notified of the thread's new location. The thread manager also stores information concerning the workstation from which the thread was launched, and the windows on that workstation to which the thread's output is to be directed.

After it has established the object's address space at a compute server, the user object manager then looks up the appropriate entry point in the object and creates an isiba to execute from that address. As the isiba executes the method, it page-faults on the pages not already resident and these are brought in from a data server.

Clouds summary and discussion

In summary, Clouds is a microkernel-based operating system for distributed objects. It is designed to support the object-thread paradigm, in which passive objects containing protected code and data are invoked by independent threads that can cross computer boundaries. Data is passed between objects only in the form of invocation parameters. Clouds provides a persistent, single-level store, so that object data is automatically saved to secondary storage and fetched from secondary storage without the programmer's intervention. Consequently, there is no requirement for a separate file construct, and objects do not require marshalling before they can be copied to secondary storage.

The Clouds system architecture consists of: workstations for users; data servers for storing the code and data segments from which objects are composed, and compute servers that execute object methods. Load balancing is carried out automatically across the compute servers as invocations take place. Since threads can execute within the same object at compute servers that do not share physical memory, the distributed shared memory abstraction is required, and support for this is allowed in the form of DSM client and server system objects, which can be configured to execute in the Ra microkernel's address space. All high-level service provision in Clouds (including network communication) is provided by the UNIX workstations and by Ra system objects.

All of the Clouds features that we have mentioned have been implemented on Sun 3/50 and Sun 3/60 computers. Null invocation timings have been measured from 8 milliseconds to 103 milliseconds, depending on whether the invoked object's data are in memory or requires transport from a data server [Dasgupta *et al*. 1991].

The Clouds invocation mechanism is interesting by virtue of its contrast with our other distributed operating system case studies. However, support for fine-grained objects is limited and the efficiency of the invocation mechanism seems questionable, since it requires virtual memory manipulations regardless of the amount of data which is required to be transferred by an invocation.

The Clouds DSM-based invocation mechanism belongs to one of three main models, the other two of which are *RPC-based invocation* and *proxies*. In RPC-based invocation, objects are implemented inside execution environments such as Mach tasks. An invoker performs a remote procedure call to a port representing the target object. A thread in the target object's host execution environment executes the method and returns any results. By migrating the object's port, the object itself can be transparently migrated between execution environments. The COOL system, constructed over the Chorus kernel sometimes uses this invocation style and sometimes uses distributed shared memory techniques similar to Clouds. COOL chooses the mechanism according to efficiency considerations.

In the third style of implementing invocations, based on so-called proxies, objects are again implemented behind ports in execution environments. However, when an invocation is performed upon a remote object, the invocation is issued transparently firstly to a local object called a proxy [Shapiro 1986]. The proxy can forward the invocation identifier and parameters in a message to the remote object's port. But, since the proxy is an object, it may alternatively use specialized techniques to implement the invocation. For example, it could satisfy the invocation with data cached locally. Or, as another example, it could multicast the invocation message to a collection of remote objects, in a scheme that uses object replication to achieve high availability. A comparison of these three techniques is given by Blair and Lea [1992].

The brief description we have given of Clouds omits some notable features that are necessary to construct full support for distributed objects in general. Some of these features are being investigated by the Clouds project. They include: object location algorithms; garbage collection; dynamically loading persistent objects into existing execution environments; object migration policies for load balancing and network traffic reduction; and the details of the DSM implementation and its interaction with concurrency control mechanisms such as locks. For further information about system support for distributed objects, the reader is referred to COOL [Lea *et al*. 1993], SOR [Shapiro 1989], Emerald [Jul *et al*. 1988] and Amber [Chase *et al*. 1989].

References

[Blair and Lea 1992]	Blair, G.S. and Lea, R. (1992). The Impact of Distribution on the Object-Oriented Approach to Software Development, *IEE/BCS Software Engineering Journal*, vol. 7, no. 2.
[Chase *et al.* 1989]	Chase, J.S., Amador, F.G., Lazowska, E.D., Levy, H.M. and Littlefield, R.J. (1989). The Amber System: Parallel Programming on a Network of Multiprocessors. *Proc. 12th. ACM Sym. on Operating System Principles*, pp. 147-58, December.
[Meyer 1988]	Meyer, B. (1988). *Object-Oriented Software Construction*. Englewood Cliffs NJ: Prentice-Hall.
[Shapiro 1986]	Shapiro, M. (1986). Structure and encapsulation in distributed systems: the proxy principle. *Proc. 6th IEEE Int. Conf. on Distributed Computing Systems*, Cambridge MA, pp.198–204.
[Shapiro 1989]	Shapiro, M. and Gautron, P. (1989). Persistence and migration for C++ Objects. *Proc. Third European Conference on Object-Oriented Programming*, Nottingham, pp. 191–203.
[Stroustrup 1986]	Stroustrup, B. (1986). *The C++ Programming Language*. Reading MA: Addison-Wesley.