Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1987

Architecture and Operation Invocation in the Clouds Kernel

Eugene H. Spafford
Purdue University, spaf@cs.purdue.edu

Report Number:
87-730

Spafford, Eugene H., "Architecture and Operation Invocation in the Clouds Kernel" (1987). *Department of Computer Science Technical Reports.* Paper 630.
https://docs.lib.purdue.edu/cstech/630

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

ARCHITECTURE AND OPERATION INVOCATION IN THE *CLOUDS* KERNEL

Eugene H. Spafford

CSD-TR-730
December 1987

Architecture and Operation Invocation in the *CLOUDS* Kernel*

Purdue University Technical Report CSD-TR-730

Eugene H. Spafford

Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907-2004

spaf@cs.purdue.edu

ABSTRACT

Many distributed operating systems have been developed in recent years based on the action/object paradigm. The *Clouds* multicomputer system provides a fault-tolerant distributed computing environment built from *passive* data objects, *fault-atomic* transactions, processes, and a global kernel interface implemented on top of unreliable hardware.

Key to the successful functioning of *Clouds* is its simple architecture consisting of passive, abstract data objects and the uniform operation invocation mechanism. This architecture allows plain processes or nested transactions to access user and system data in a transparent, uniform manner, whether those objects are local to the current machine or on some remote processor. The same basic interface used to make operation invocation requests on objects can be used to spawn processes and actions, and to gain access to restricted kernel services.

This paper presents an abbreviated description of the *Clouds* architecture and its relation to the operation of the invocation mechanism, including remote invocation, per-object access control, and location independent invocation. Some conclusions derived from the first prototype are also presented.

December 18, 1987

* Portions of this work were funded at Georgia Institute of Technology by NSF grant DCR-8316590 and NASA grant NAG-1-430.

Architecture and Operation Invocation in the *CLOUDS* Kernel*

Purdue University Technical Report CSD-TR-730

Eugene H. Spafford

Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907-2004

spaf@cs.purdue.edu

1. Background

Recently a great deal of research has been focused on the potential benefits of distributed systems. Distributed system offer the potential of a fault-tolerant computing environment by replication and redundancy. A distributed system also suggests increased computing power through the combination and application of resources. Multiple machines, however, raise many questions relating to communication, consistency, reliability, configuration, and user interfaces.

The *Clouds* project began in 1982 with an examination of how to construct and apply a useful distributed system that could address these concerns yet be built on general-purpose, "off-the-shelf," potentially unreliable hardware. The approach was influenced by the concept of a *fully distributed processing system*:[Ensl78] a system (partially) characterized by the lack of any central locus of control or authority. There is no central scheduler, name server, or other single entity which must be available for the components of the system to operate. Such a system allows autonomous operation of individual nodes should the need arise, due to failure or due to administrative fiat. This kind of system is appealing from a number of standpoints, not least of which is its inherent potential tolerance of (at least) single-point failures.[1] In systems without such full distribution, the failure of a component supporting a centralized service, such as a name server, results in the failure of the entire system until that service is restored or replaced. An explicit goal of the *Clouds* design has been to construct a system that will tolerate and recover from single-point failures.

Preliminary design revealed a few requirements as basic to this paradigm. First, for a distributed system to provide general utility, it would be necessary to support the synchronized shared access and interaction of distributed items of data of arbitrary type and size. Secondly, those items of data needed to stay consistent across failures, and that consistency had to be made automatic in some way (further, there could be no central authority to provide and enforce such consistency since such a central authority could also fail). These requirements are also basic to many other distributed operating system projects, and the resulting *Clouds* view was not an isolated one: an object/transaction paradigm is a good way to structure a distributed operating system. Objects present a convenient means of abstracting and isolating data, and transactions provide an abstraction to use to keep that data consistent in the face of failures. Examples of related approaches include Eden,[Jeas82, Almc83, Noe85] ISIS,[Birm85, Jose85] Cosmos,[Nico85] and Argus.[Lisk83, Weih83]

The use of objects and transactions in *Clouds* is intended to be at a lower level than any of these other systems, however. In any operating system, many things depend upon data that must be kept consistent across failures (e.g., directories, scheduler queues, accounting information). The *Clouds* philosophy is that if objects and transactions are implemented at the lowest possible level (in the kernel), they can then be used to build the remainder of the reliable, distributed operating system itself, as well construct needed

[1] A *single-point* failure, in the context of *Clouds*, is a failure which affects just a single component within any arbitrary period of time and which is detectable. All such failures are considered to be *fail-stop* or *halting* failures in that each component will operate correctly or not at all.

applications. This was one of the major goals of the overall *Clouds* project, and is key to understanding the resulting architecture.[McKe83,McKe83........,Allc83] Many of these features of *Clouds* have been further validated by adaption and use in the *Alpha* system.[Nort87]

2. System Architecture

A *Clouds* multicomputer consists of four types of logical entities: objects, actions, processes, and sub-kernels. Sub-kernels actually represent the replicated kernel—the virtual *Clouds* machine—but we will mention them here since it is the sub-kernels that are responsible for implementing the interface to operation invocations in *Clouds*. We will describe the major features of these entities in the following sections; readers desiring an in-depth understanding of the overall design should consult[Allc83, Spaf86], and[Wilk87].

2.1. Objects

The *Clouds* architecture uses large and medium-grained abstract data objects as a means of encapsulating functionality and isolating errors and recovery considerations. These objects, when programmed correctly, provide excellent shared access to the data they encapsulate because locking and synchronization of their internal data can be tailored according to the semantics of that data.

Clouds objects are abstract data items of varying size and complexity, defined along with the operations possible on those structures. These operations may include explicit synchronization and recovery operations, exception handlers, and dynamic storage management code. No other user access may be made to the data within an object except through these operations.[2]

Typical *Clouds* objects might include items like output objects in a printer or plotter queue, text file objects, and user mailbox objects, although it is possible to define and support objects consisting of single integers or characters. The object space is flat, with no contained objects or explicit type inheritance. Objects may not span machine boundaries.[3]

Unlike the *active* objects used in other research approaches (e.g., Eden's *Ejects* or Argus' *Guardians*), *Clouds* objects are *passive*—there are no processes bound to the object instances. All activity within *Clouds* objects is as a result of an external process or action doing an operation invocation on the object—entering the object's code and data space to perform a defined operation, and then leaving that address space when the operation is completed. This paradigm means that there are no specific processes associated with any object on a long-term basis, nor is there any long-term process management associated with each object.

Passive objects have at least these distinct advantages compared to active objects:

- Passive objects are simpler to code and support (especially when implemented on a "bare" machine) since they require no explicit code to support processes within them. For instance, passive objects can be smaller than active objects, and need suffer fewer (if any) restrictions about being paged in and out of memory at any time. Furthermore, local invocations of passive objects do not require process context switches, as do active objects.
- The passive object operation concept corresponds more closely with a paradigm familiar to most users—that of the procedure call. Remote operations map into *remote procedure calls (RPC)* in a natural manner as procedure calls. RPC operations mapped onto active objects require the inclusion of *ports* or *rendezvous* constructs which may not be as easily understood by programmers.
- There is no limit on the concurrency possible within the code of a passive object other than what is specifically designed into that code. Active objects generally provide a limited number of servers and therefore activity within those objects is limited to the number of servers

[2] Kernel operations for transaction and memory management (paging)[Pitt86] and debugging[Lin87] are special cases controlled by the kernel.

[3] More complete semantics and structural details are given in works on programming in *Clouds*.[Aham87,LeBl85,Wilk86,Wilk87]

available. Increased concurrency within a passive object does not necessitate increasing its size, since no process context is ever stored within the object.

- *Clouds* objects may be easily moved, deleted or replaced at almost any time; active objects require interaction with the kernel to save process states, instantiate new server processes, etc.

Further, by allowing the kernel to view *Clouds* objects as similar, abstract entities, it is possible to design common operations which can operate on objects of any type. Such operations include copying or cloning of the objects to provide increased availability, and migration of objects to other machines. Coupled with the fact that each object defines its own access and recovery, *Clouds* objects fit in perfectly with the autonomous nature of our definition of a fully distributed system.

2.2. Processes

Processes are the basic unit of activity in the *Clouds* system. A *Clouds* process is conceptually similar to the traditional notion of a process or thread, in that it represents a series of related activities over a period of time. However, *Clouds* processes do not have a distinct address space associated with them. A process consists only of its process control block and registers. As a process executes, it enters the memory context of *Clouds* objects and executes the code therein on the data within those objects. When it exits the object (returns), its context is replaced by the context of another object, or by the context of the machine sub-kernel.

Processes are created by calls on the *Clouds* kernel specifying an initial object context. The process is given a PCB and is mapped into the context of the specified object. Along with that mapping, the process is provided with a temporary stack associated with that object. Should the process invoke an operation on another object, its stack and context in the first object are unmapped and saved, and it is mapped into the called object along with a new stack. Upon return, the second object is unmapped and the stack discarded, and the saved contexts are remapped. Argument transfer between calls and process creation are both described later in this document.

2.3. Actions

Clouds actions are similar to the more traditional notion of a transaction,[Moss81] but they implement *failure atomicity* and not necessarily *view atomicity*.[Alle83] That is, there is nothing inherent in the semantics of *Clouds* transactions that prevent them from observing the effects of other actions, but all-or-nothing behavior in the face of failure or explicit abort is preserved. To achieve serializability with *Clouds* actions, the programmer must provide explicit synchronization using *Clouds* primitives designed for that purpose.[McKe85]

Actions are implemented as *Clouds* processes that are specially tracked as they execute. Whenever a *Clouds* action invokes an operation on an object marked as recoverable (i.e., intended to be failure-atomic), the kernel first invokes an *action manager* object to record the invocation. If the recoverable object is changed by the action, the changes are made in a temporary manner.[4] When the action completes, it either *commits* or *aborts*. If the action commits, all of its changes to recoverable objects are written to permanent storage. If the action aborts, all of its changes are ignored and made to appear as if they had never occurred. Machine or software failures cause actions to abort if they had accessed any objects related to the failure.

Clouds objects are marked as recoverable or not, and only actions may invoke operations on recoverable objects. Processes are not allowed to perform commit or abort operations, either.[5] In all other respects, actions and processes are identical.

Actually, the design of *Clouds* objects allows many types of resilient object to exist, along with matching forms of actions. In fact, multiple types could coexist in the same system and potentially call each other in a transparent manner, with the kernel determining the action type based on the object type

[4] The first *Clouds* prototype uses object shadowing[Gray81, Pitt86] as its primary recovery mechanism, although other methods, including logging and version stacks, have been contemplated.

[5] Actions may invoke operations on non-recoverable (plain) objects, too, but their effects are not undone in the case of failure or abort.

information; this has not yet been explored in the prototype, nor in associated language work, but the potential is present in the design.

If the action manager allows them, *subactions* are also supported. The effects of subactions in relation to their ancestors and siblings is determined by the action manager involved. The *Clouds* prototype may eventually support subactions, with semantics as defined in[Wilk87] and [Kenl86].

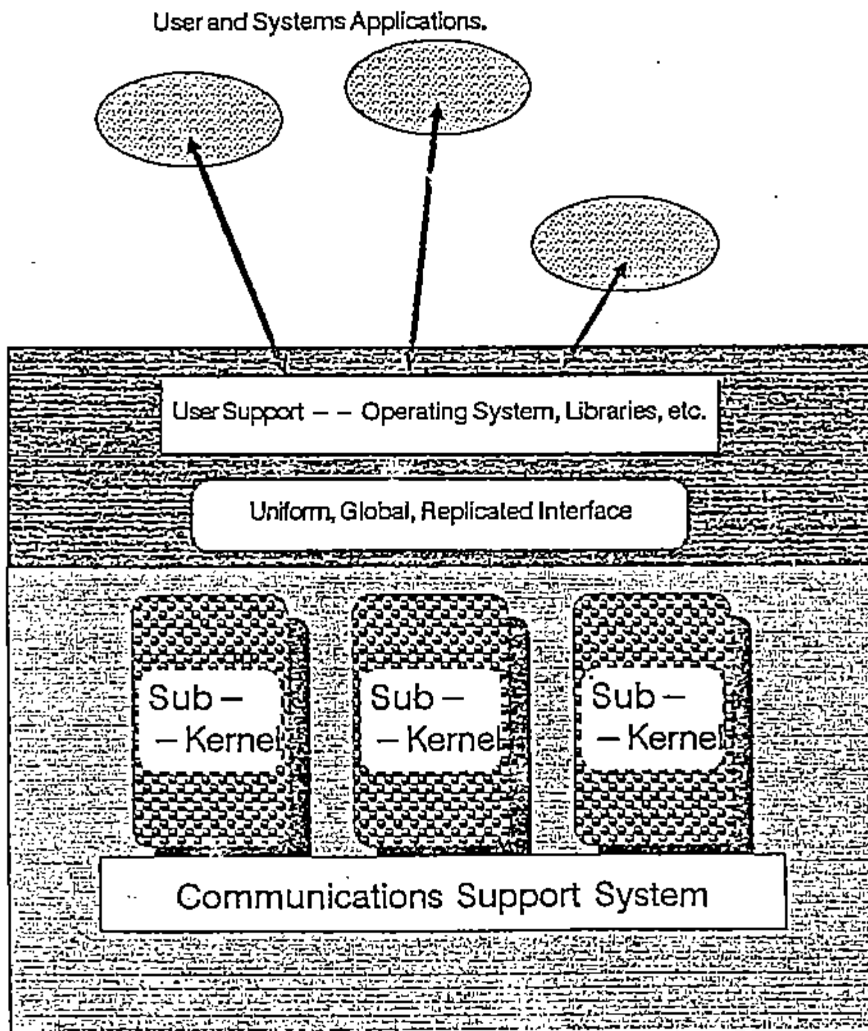

Figure 1: Logical Structure of a *Clouds* Multicomputer

2.4. The Sub-kernels

To a user, a *Clouds* multicomputer is one single computer that provides persistent abstract data objects, processes and actions. This view is the same whether the multicomputer is composed of a single machine or a large number of machines. To achieve this view, *Clouds* implements a global operating system and a global interface on replicated local kernels or *sub-kernels* (figure 1). Object operation invocations occur in this global space of objects and kernel interface.

Each machine supports a *Clouds* sub-kernel. Each sub-kernel is responsible for providing an identical set of virtual machine services and functions[6] which can be referenced by the objects and by the global

[6] These include I/O, paging, and process control operations. Specifics are detailed in [Spaf86].

operating system. Each sub-kernel also supports an instance of the kernel interface through which applications may make requests of kernel and operating system services (figure 2).

When a user wishes to develop a new application using *Clouds* objects, s/he first programs the application in an object-oriented language such as *Aeolus*.[Wilk85] The application is programmed as one or more abstract data objects and as the operations on those objects. The user may also include implicit or explicit support of certain kinds of recovery operations and synchronization operations in the object definition. Recovery operations can be used to provide a certain amount of fault tolerance in case of failure, and the synchronization allows the user to regulate shared accesses to the object. The user is never forced to include any of these operations, however, and may program a simple non-recoverable, non-synchronized object if desired.

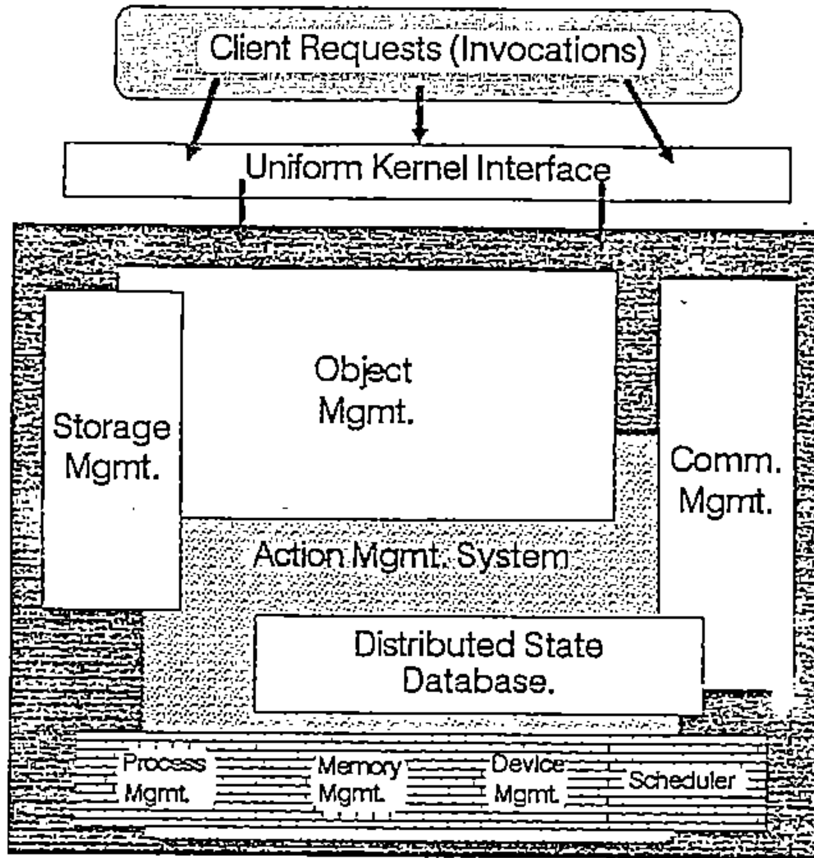

Figure 2: Structure of a *Clouds* Sub-kernel

All references to objects are done without knowledge of the location and accessibility of the target objects, and the sub-kernels cooperate to bind the references to the current location of the indicated object. This is done through *search-and-invoke* operations when the needed objects are not found on the local machine. Knowledge of where objects were located earlier is kept only as hints, and thus objects can be moved without notifying any central authority or without disrupting future references.

3. The Invocation Mechanism

3.1. Naming

Every object, process, action and sub-kernel is uniquely named by a *sysname*. A sysname is constructed from a timestamp, birthplace and type parameter, and is guaranteed unique (no two items will be referenced by the same sysname). The type field provides a hint about the type of the item referenced—whether it is a process, an action, a user-defined object, etc. In some cases it is possible for two sysnames to have the same identifier portion, but different type fields. This occurs when two different forms of access are allowed.[7]

A *Clouds* sysname coupled with an access structure comprises a *Clouds* capability. Capabilities are employed by user code to reference kernel operations, objects, processes and actions. The access rights field is treated as an untyped bit string; the kernel does not define the meaning of the bits, nor does it determine or enforce access rights on anything other than its own operations. Rather, it provides the bit string to the object being invoked for a per-invocation validation check. The programmer of the object may elect to have no check performed, or s/he may require some authorization and use the bit string for this.

Sysnames are not available to user code, but are used within privileged code to locate and operate on various items. Sysnames are created as needed by the kernel. Capabilities cannot be altered or created by user code, but can be obtained and modified through calls on the sub-kernel. Capabilities can be passed as parameters in invocations; this is the only means of referencing an external object during an object operation invocation.[8]

3.2. Object Structures

Every *Clouds* object has a defining structure associated with it. This structure contains fields indicating memory mapping and protection information about the object, type information, recovery information, and accounting information. Whenever an object is referenced, the storage version of this object descriptor is brought into memory (if not already present) and various fields are initialized. The in-memory version of this structure is known as the *object control block* or *OCB*.

Each sub-kernel maintains a cache of currently active and recently used OCBs. This cache, the *active object table* or *AOT*, can be used to quickly find the OCB for a referenced object. References through sysnames (and capabilities) are first checked against the AOT. Entries not found indicate a reference to an object not currently in memory, or an object not present on this machine, or possibly an object which no longer exists due to removal or failure.

3.3. Invocation

Code inside any object or sub-kernel can make an object operation invocation. Before such an operation is requested, the invoking application code must collect and format the arguments to the invocation into structures known as *arglists*. Arglists describe the parameters to the call, including number and size (but not type). Arglists are manipulated by the sub-kernel during the invocation to enforce value-result semantics, and to guard against result overflow in size or number.

The interpretation of the data within arglists is determined by the code at either end of the invocation, and is otherwise treated as untyped bytes; operations to format data into a canonical representation could be performed by the objects involved, if necessary, for use in a heterogeneous machine environment. As elsewhere in the *Clouds* design, this kind of decision is left to the programmer and application rather than forced by the underlying mechanism.

Once the arglists are built, the application code makes the invocation request on the local sub-kernel with four parameters: a capability to the object being sought, an operation number to be performed, and the two arglists (one for input, one for output). These arguments are copied into a new stack for the process so as to be available to the invoked object after the current object is unmapped.

---

[7] As an example, a process can also be considered as an object. That object can have read and write operations defined on it so as to allow a debugger to control the associated process through simple object calls examining the registers and stack, in a manner similar to [Kill84].

[8] The reason for this is explained in [Alle83] and is necessary to preserve atomicity constraints in operations crossing machine boundaries without requiring every such RPC to be a separate subaction.

All operations requested by users of the *Clouds* system are mapped into calls to the *kernel interface*. Conceptually, the kernel interface extends across machine boundaries and exists on each sub-kernel within the *Clouds* system. In actuality, the interface is replicated on each machine. The kernel interface implements the *invoke operation* and *return from invocation* operation. The four parameters for an invocation are passed to the first of these two operations.

The first step in the invocation occurs as the kernel interface finds where the invocation will occur. This is determined by the location of the actual target object instance. The first check is try to find in the AOT the sysname portion of the target capability. If no match is found, the sysname is checked against the *maybe table*.[Pitt86]

The maybe table is an approximate membership tester, like a Bloom filter,[Bloo70] containing information about all stored and active objects present on this machine. The table provides a quick determination of whether or not an object is present on the local machine. A negative response is always definitive, but a positive response simply means the object might be present. If the maybe table indicates that the searched-for object might be present locally, a search is performed on the secondary storage directories in an attempt to locate the object.

If the named object is local and already represented in the AOT, then invocation proceeds directly (described below). Otherwise, if the object is local to this machine but not in the AOT, then a call is made on internal routines to map the object into memory from secondary storage and proceed with the invocation.

3.3.1. Failures

All failures of invocation requests are returned to the caller as a single failure code. There is no distinction made as to the reason of the failure since it is not always possible to determine the actual reason for each failure: a *not found* condition could mean the object does not exist, the object has been deleted, the network is partitioned, the disk is not mounted, etc. Furthermore, it is a potential security hole to distinguish between different failure like "access disallowed" and "object not found." Actions and subactions attempting an invocation automatically abort on failure, so as to preserve *exactly-once* semantics. Processes may retry on failure or take alternate actions, as the programmer decides.

3.3.2. Local Invocation

The operation number provided in the invocation request is checked for validity. In the prototype, if the number is positive it indicates a regular operation (programmer defined), and is compared against a range field in the OCB. If the operation number is greater than this limit, the operation is disallowed (no such operation) and an invocation failure is signalled. If the supplied operation number is negative, it indicates a special operation, such as *abort* or *commit*, and fields in the OCB are checked to ensure that the operation is actually defined and that the object is recoverable. At the same time the object type is checked to see if it is recoverable, the caller is checked to see if it is an action. If the caller is not and the object is recoverable, the operation is disallowed and a failure signalled; only actions are allowed to access recoverable objects.[9]

Next, a record is linked to the current PCB containing context information about the state and location of the current object-space stack, current virtual memory mapping parameters, and a pointer to the client-supplied output parameter list. Then, the current object-space stack is made inaccessible, a new stack is allocated and initialized, and the memory mapping registers are loaded to enable reference of the new "current" object. As a result, the memory context of the invoked object (and only that object) is now accessible to the process.

Finally, some form of branch or subroutine call (as appropriate for the machine) is made to a common starting address in the object, as given in the OCB. Provided as arguments to the call are the sysname of the invoking object, the access rights portion of the capability used to invoke this operation, the operation number, and a pointer to the new input argument list.

[9] However, actions may freely access nonrecoverable objects. When such accesses occur, they are treated as if the actions were plain processes.

If the object is recoverable, instead of performing the branch or call directly, a call is made on the action management subsystem with the arguments and the address of the entrypoint. After determining the validity of the reference and resolving any lock and visibility considerations, the action management code will perform the branch or call with the given arguments.

The code within the entry routine of the object performs any necessary validation and synchronization specific to the object, based on the information provided by the sub-kernel interface, and then a branch is made to the code necessary to execute the indicated operation. Note that since the call provides both the access rights word and the sysname of the invoking object, it is possible to perform highly-specific access and locking operations on an object-by-object basis.

3.3.3. Return

When the execution within the object is complete, the code will execute an *object return* operation on the sub-kernel, supplying as arguments a success/failure flag and a pointer to an arglist describing the result parameters. If the returned status indicates success, the kernel interface code transfers the output values in the arglist provided by the return call into the locations specified in the results arglist supplied at the time of invocation. The count and overflow fields in that arglist are set appropriately as part of this transfer. If more return arguments are provided than the user allocated room for, overflow flags are set. If the returned status indicates failure, then no values are transferred (they are assumed to be invalid) and the count field of the user-supplied arglist is set to zero to reflect that fact.

Next, the saved object context information is unlinked from the PCB, and its values restored, effectively resuming the state the process was in prior to the invocation request. This includes updating the virtual memory registers and restoring the previous user stack. This also has the effect of unmapping the invoked object and remapping the invoking object.

If the current process is acting under the auspices of an action, and if the type field in the OCB of the object being returned from indicates that the object is recoverable, control is passed to action management code. In this specific case, a returned status flag showing failure indicates an *abort*.[10] Control is finally returned to the calling code. The value returned by the *invoke* function is the value of the status code provided in the object return call.

---

[10] On the other hand, a returned status of "success" does not necessarily indicate *commit*. That is something which must be performed with a separate invocation on the action management object.

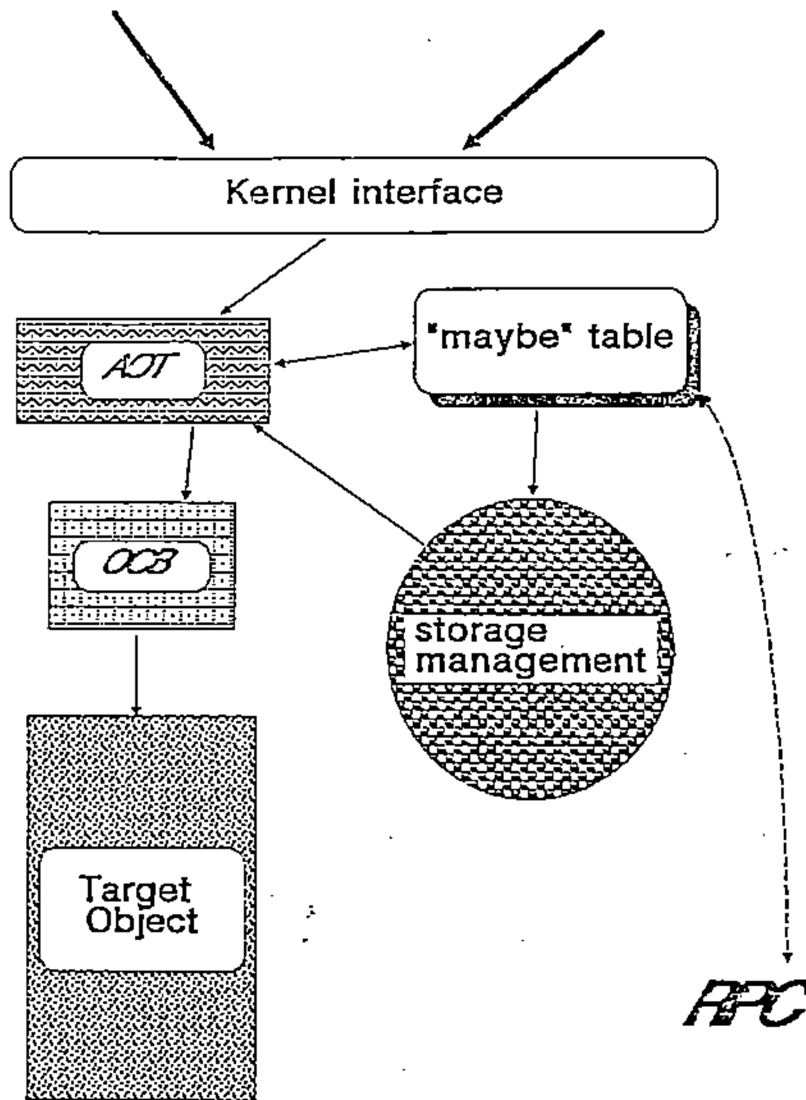

Figure 3: Object Operation Invocation

3.3.4. Remote Invocation

If the capability provided references an object which is not found locally, it is assumed to be available through one of the other sub-kernels on another machine. As such, the kernel interface constructs an RPC *search-and-invoke* request containing the provided capability and operation number, the input argument list, the sysname of the current process, and the action status of the invoking process. This RPC is then broadcast as a *search and invoke* operation to the other machines.[11]

[11] Methods other than simple broadcast or multicast are used, but the effect is similar. The network topology obviously plays a role here, as does the requirement that no single failure at an intermediate node prevent the call from succeeding. The references provide more specific details. This mechanism has not yet been fully developed and tested for a large-scale system.

If no reply comes within an implementation-dependent timeout period, a failure is signalled. If the RPC produces an acknowledgment that the invocation has started, the caller may elect to set a timer to wait for some longer period of time for results to be returned.

Remotely, code on each machine receiving the RPC checks the object name specified against the local maybe table. If the result shows that the object cannot be present locally, the RPC request is ignored. Otherwise, a slave process is given the arguments and dispatched to try the invocation locally on behalf of the invoking process. If the object is located locally, the slave process will succeed with its attempt at invocation, and the operation will be performed. If the object is not located locally, the slave simply terminates. Since all capabilities refer to unique objects, it is clear that only one slave process in the entire multicomputer will ever succeed in the invocation attempt.

When the object code completes on the remote node, the interface code handling the return request will note that the work was being done on behalf of a remote invocation. Instead of copying the return arguments into the caller's address space (that does not exist on that remote machine), it will instead format the return values and status code into an RPC reply message that will be sent back to the calling machine's kernel interface. Upon receiving the return arguments, the kernel interface will process the returned arguments as if they had been generated locally.

3.3.5. Processes and Actions

The mechanism for creating new processes and actions (and subactions) is almost identical to object operation invocation—in fact, the creation calls on the kernel look exactly like an object operation invocation. The parameters supplied to the kernel interface in these cases include a capability to an object, an operation number, and arglist structures. The newly created process or action starts execution by invoking the operation indicated on the object named by the provided capability using the provided input arguments. From the standpoint of the invoked object, it appears exactly like a regular invocation.

From the standpoint of the new process, its "birth" appears exactly like an invocation, except there is no previous context to restore, and there are no parameters to return. From the standpoint of the "parent" process, the spawning of the child appears as an object invocation; the return values include only process id information. The parent can query the status of its children at any time with calls to the subkernel using a capability to the child process.

This mechanism allows easy, consistent access to the process and action facilities. Code can be executed as (effectively) object subroutines, or executed by separate processes or actions. The same code can be executed by top-level actions or subactions with absolutely no change in the (called) code.

This approach to process creation is not completely new (cf. Mesa[Lamp80]), but it is somewhat unique in that there is no corresponding *join* nor is there any copying of parent context done. Additionally, actions and subactions can use the exact same mechanism.

3.3.6. Access to Kernel Services

The kernel interface implements a pseudo-object representing the kernel. Users may make invocations on this object to spawn processes, get accounting information, access synchronization primitives, and access other typical services. Processes can also make invocations on more privileged services, such as direct access to device drivers, assuming they have proper capabilities. Thus, special services may be established for particular users who have been given the authority to access them. Furthermore, this also allows new OS services, such as file servers and schedulers, to be written outside the kernel—they only need access to the restricted kernel operations necessary for their operation. These external services use the same interface as all other objects, they reside "outside" the kernel, and multiple versions can coexist, if desired. This mechanism is extremely flexible and can be tailored to suit almost any configuration of services desired.

Since the *Clouds* kernel has been carefully designed to implement only mechanism without built-in policy whenever possible, it is fairly simple to write traditional operating system functions such as file systems and schedulers as objects outside the kernel; in fact, this is one of the longer-term goals behind the overall *Clouds* architecture. Furthermore the passive object approach allows object-context switching to occur fast enough to code time-critical OS functions in these objects without worrying about the overhead

introduced by process context switching.

4. Current Status

A prototype *Clouds* kernel was completed on a group of VAX 11/750™ computers in early 1986, after approximately three man-years of effort. The prototype supported objects, processes, action tracking (but not the full action mechanism described in[Kenl86]), RPC, disk support, intermachine communication, debugging and trace code, and the entire invocation mechanism described in this document. The total code (with comments) required to support this kernel was under 12,000 lines of C, and a few hundred lines of assembler. We view the compactness of this sub-kernel as another strong argument in favor of passive objects.

The unoptimized version of the invocation mechanism described here has been measured as taking 4.39 milliseconds per invocation and return for objects local to the current machine and present in the AOT. A simple procedure call and return on the same machine takes 40μsec. The majority of the object invocation time is taken by the switch in memory contexts caused by entering and leaving object spaces, and by copying of the arglists so as to present only value parameters. Unfortunately, the memory system on a VAX 750 is such that the only way to easily achieve a memory context switch for a multi-page object is to force a process context switch, thus negating some of our anticipated benefits for passive objects.

Initial measurements of invocation across machine boundaries has been overshadowed by timing delays in the Ethernet driver. The reason for these delays has not been determined, but leads to remote invocation times of approximately 40 msec, although some trials have resulted in times of less than 25 msec; published figures for hardware and software timing figures of related systems suggest that these times could eventually be reduced by approximately an order of magnitude, although we are unsure of the level of development effort required to achieve such a speed-up.

User and OS-level objects have been written to exercise the kernel services, and provide needed services. A TCP/IP mechanism was added to the kernel in late 1986, and remote interactive debugging and monitoring facilities were developed on workstations running UNIX®. The current Clouds team is planning a complete rewrite on different hardware with a somewhat modified design[12] as part of a NSF CER project, awarded in 1987. Apparently, that effort will derive little from the experiences with the first prototype, and may well result in a redefinition of some of the basic goals driving the original *Clouds* design.[Bern87] This author is also contemplating a (somewhat different) implementation of the original design on another hardware base with a focus on a more flexible memory architecture and a multiprocessor-oriented design, along with more explicit support for security and a different network mechanism.

5. Conclusions

An initial implementation of a *Clouds* kernel has been done on top of the "bare hardware" of a machine rather than attempt to prototype it on top of an existing system, such as UNIX, to ensure the availability of needed constructs at a primitive enough level. The use of passive objects as the basis for the *Clouds* design resulted in a small, compact kernel with considerable flexibility, and a distributed environment presenting programmers with considerable freedom of design.

The operation invocation mechanism allows a *Clouds* programmer to employ the same basic methods when operating on processes, recoverable objects, or the kernel itself. This makes programming simpler, and increases in simplicity usually enhance accuracy and reliability. The mechanism (and *Clouds* philosophy) allow the user to choose the representation and level of support appropriate to the needs at hand.

The *Clouds* mechanism for handling operation invocations is also significant because of restrictions not present in its design. To add new operations or change access policies on an existing object it is not necessary to invalidate or modify existing references to that object. Instead, a modified version of the

---

™ VAX and VMS are trademarks of the Digital Equipment Corporation.

® UNIX is a registered trademark of AT&T.

[12] As discussed by P. Hutto at the Works in Progress session, 1987 SOSP, Austin.

object supporting at least the same operations as before is created with the same sysname as the previous version. When invoked with one of the "old" capabilities, the invocation mechanism finds the modified object during the search-and-invoke phase and the operations are performed (assuming the new access rights within the object are met). A "new" capability to the object will also work correctly, referencing any new operations defined. Both kinds of change work because the kernel only binds references to objects, performs memory mapping operations, and transfers arguments—it does not actually manage objects or meddle with their operational interfaces.

Experience with the *Clouds* prototype has shown the mechanism to be quite feasible and potentially efficient, given minimal hardware support. Research is currently underway to determine the full potential of the mechanism in something other than a test implementation of *Clouds*. Those results will be reported at a later date.

References

Aham87.
Ahamad, M., P. Dasgupta, R. J. LeBlanc, and C. T. Wilkes, "Fault-Tolerant Computing in Object Based Distributed Operating Systems," *PROCEEDINGS OF THE SIXTH SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, pp. 115-125, IEEE Computer Society, Williamsburg, VA, March 1987.

Allc83.Allchin, J. E., "An Architecture for Reliable Decentralized Systems," PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1983. Also released as technical report GIT-ICS-83/23

Alme83.
Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," TECHNICAL REPORT 83-10-05, University of Washington Department of Computer Science, October 1983.

Bern87.
Bernabeu, J., P. Hutto, and Y. Khaldi, "A Modest Proposal," DISTRIBUTED SYSTEMS GROUP MEMO GIT-ICS-DSG-87/01, Georgia Institute of Technology, Atlanta, GA, 18 June 1987.

Birm85.
Birman, K. P. and others, "An Overview of the ISIS Project," *DISTRIBUTED PROCESSING TECHNICAL COMMITTEE NEWSLETTER*, vol. 7, no. 2, IEEE Computer Society, October 1985. Special issue on Reliable Distributed Systems

Bloo70.
Bloom, B. H., "Space/Time Trade-offs in Hash Coding with Allowable Errors," *COMMUNICATIONS OF THE ACM*, vol. 7, no. 13, pp. 422-426, July, 1970.

Ensl78.
Enslow, P. H., "What is a "Distributed" Processing System?," *COMPUTER*, vol. 11, no. 1, pp. 13-21, IEEE, January 1978.

Gray81.
Gray, J. N. and others, "The Recovery Manager of the System R Database Manager," *COMPUTING SURVEYS*, vol. 13, no. 2, ACM, June 1981.

Jeas82.Jeasop, W. H., J. D. Noe, D. M. Jacobson, J. Baer, and C. Pu, "The Eden Transaction-Based File System," *PROCEEDINGS OF THE SECOND SYMPOSIUM ON RELIABILITY IN DISTRIBUTED SOFTWARE AND DATABASE SYSTEMS*, pp. 163-169, IEEE, Pittsburgh, PA, July 1982.

Jose85.Joseph, T. A., "Low-Cost Management of Replicated Data," PH.D. DISS., Department of Computer Science, Cornell University, Ithaca, NY, November 1985. Also released as Technical Report TR 85-712

Kenl86.
Kenley, G. G., "An Action Management System for a Distributed Operating System," M.S. THESIS, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. Also released as technical report GIT-ICS-86/01

Kill84.Killian, T., "Processes as Files in Eighth Edition Unix," *PROCEEDINGS OF THE SUMMER USENIX CONFERENCE*, Usenix Association, Salt Lake City, Utah, July 1984.

Lamp80.
Lampson, B. W. and D. D. Redell, "Experience with Processes and Monitors in Mesa," *COMMUNICATIONS OF THE ACM*, vol. 23, no. 2, pp. 105-117, February 1980.

LeBl85.
LeBlanc, R. J. and C. T. Wilkes, "Systems Programming with Objects and Actions," *PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS*, Denver, July 1985. Also released, in expanded form, as technical report GIT-ICS-85/03

Lin87.Lin, C., "The Design of a Distributed Debugger for Action-Based Object-Oriented Programs," PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

Lisk83.
Liskov, B. and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust Distributed Programs," *TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, vol. 5, no. 3, ACM, July 1983.

McKe83.
McKendry, M. S., J. E. Allchin, and W. C. Thibault, "Architecture for a Global Operating System," *IEEE INFOCOM*, April 1983.

McKe83.
McKendry, M. S. and J. E. Allchin, "Synchronization and Recovery of Actions," *PROCEEDINGS OF THE SECOND SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING*, ACM SIGACT/SIGOPS, Montreal, August 1983.

McKe85.
McKendry, M. S., "Ordering Actions for Visibility," *TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 11, no. 6, IEEE, June 1985. Also released as technical report GIT-ICS-84/05

Moss81.
Moss, J., "Nested Transactions: An Approach to Reliable Distributed Computing," TECHNICAL REPORT MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

Nico85.
Nicol, J. R., G. S. Blair, and W. D. Shepherd, "A Tailored Kernel Design for a Distributed Operating System," TECHNICAL REPORT, Department of Computing, University of Lancaster, Lancaster, England, 1985.

Noe85.Noe, J. D., A. B. Proudfoot, and C. Pu, "Replication in Distributed Systems: The Eden Experience," TECHNICAL REPORT TR-85-08-06, Department of Computer Science, University of Washington, Seattle WA, September 1985.

Nort87.
Northcutt, J. D., *Mechanics for Reliable Distributed Real-Time Operating Systems*, PERSPECTIVES IN COMPUTING, 16, Academic Press, 1987.

Pitt86.Pitts, D. V., "Storage Management for a Reliable Decentralized Operating System," PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. Also released as Technical Report GIT-ICS-86/21

Spaf86.
Spafford, E. H., "Kernel Structures for a Distributed Operating System," PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1986. Also released as technical report GIT-ICS-86/16

Weih83.
Weihl, W. and B. Liskov, "Specification and Implementation of Resilient Atomic Data Types," *SYMPOSIUM ON PROGRAMMING LANGUAGE ISSUES IN SOFTWARE SYSTEMS*, June 1983.

Wilk85.
Wilkes, C. T., "Preliminary Aeolus Reference Manual," TECHNICAL REPORT GIT-ICS-85/07, School

of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985. Last Revision: 17 March 1986

Wilk86.

Wilkes, C. T. and R. J. LeBlanc, "Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System," *PROCEEDINGS OF THE 1986 INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES*, pp. 107-122, IEEE Computer Society, Miami, FL, October 1986. Also available as Technical Report GIT-ICS-86/12

Wilk87.

Wilkes, C. T., "Programming Methodologies for Resilience and Availability," PH.D. DISS., School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.