Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

S.J. Mullender

Process management in a distributed operating system

Computer Science/Department of Algorithmics & Architecture Report CS-R8713 March

69D50, 69C24, 69D25

Process Management in a Distributed Operating System

Sape J. Mullender
Centre for Mathematics & Computer Science
Amsterdam
and
DEC Systems Research Center
Palo Alto

The Distributed Systems Group at the Centre for Mathematics and Computer Science and the Vrije Universiteit in Amsterdam has designed a collection of services for the management of processes in the Amoeba distributed operating system. With a small set of kernel operations, it is possible to download, debug, migrate, and checkpoint processes.

First, the basic kernel mechanisms are described, followed by the description of a number of supporting user-space services. The paper ends with a discussion of the properties of Amoeba that made this design possible.

CR Categories: D.4, C.2.4, D.2.5.
Key Words & Phrases: Distributed operating system, process management, migration, Amoeba.
Note: This report will be submitted for publication elsewhere.

1. INTRODUCTION

As part of designing and building the Amoeba distributed operating system, we have come up with a simple set of mechanisms for process management that allows downloading, process migration, checkpointing, remote debugging and emulation of alien operating system interfaces.

The basic process management facilities are realized by two services, a kernel-space service, Kernel Service, and a user-space service, Process Service. These facilities can be augmented by other user-space services: Debug Service, Load-Balancing Service, Unix-Emulation Service, Checkpoint Service, etc.

The Amoeba Kernel can produce a representation of the state of a process which can be given to another Kernel where it is accepted for continued execution. This state consists of the memory contents in the form of a collection of segments, and a *Process Descriptor* which contains the additional state, program counters, stack pointers, system call state, etc.

Careful separation of mechanism and policy has resulted in a compact set of Kernel operations for process creation and management. A collection of user-space services provides process management policies and a simple interface for application programs.

In this paper we shall describe the mechanisms as they are being implemented in the Amoeba Distributed System at the Centre for Mathematics and Computer Science in Amsterdam. We believe that the mechanisms described here can also apply to other distributed systems.

Report CS-R8713
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

2. Processes in Distributed Systems

Our goal in designing the process management primitives described in this paper was to provide mechanisms that can do what process management primitives in existing general-purpose operating systems can do and more. The added functionality has to do with the properties of the kinds of distributed systems we are interested in: personal workstations, shared server machines and *guest systems*, connected by a fast local-area network.

The workstations are normally used by a single person, but, when nobody is using them, they are available as a computing resource to users of other workstations. The shared server machines provide a distributed file system, name service, gateways to the internet, access to printers, tape drives and other devices, etc. By 'guest systems' we mean traditional operating systems that have become connected to the distributed system with some software to allow the sharing of software between the 'new' and the 'old' world. In the case of our system, Unix† systems are still used because of the enormous body of software available to us there; software that is only slowly replaced by equivalent or better in the distributed system.

We are building a general-purpose distributed system, so the programming environment we design for is a heterogeneous one: many languages, several file systems, existing software developed on other systems, possibly a wide variety of hardware and different kinds of networks to connect the machines. The designed process abstraction must allow running existing software. There must thus be support for heavy-weight processes and emulation of foreign operating system interfaces (preferably providing binary compatibility: binaries from the foreign system must run without modification).

In this environment, sufficient protection mechanisms must be implemented to prevent one user's programs from disturbing another's. Programs from different users will frequently share one physical processor, so they must run in separate address spaces.

Not all machines can be expected to have a local file system, so programs will have to be downloaded over the network. The mechanisms that do this must be fast; some programs are several megabytes in size, so loading takes seconds, even in the best of cases, and the user is often impatiently waiting at the terminal.

Distributed applications will rely heavily on fast interprocess communication. In many distributed systems, the basic communication mechanism is the *message transaction*, a message pair: a request message from a client process to a server, followed by a reply message from the server back to the client. On top, *remote procedure call* is often provided. When carefully designed and implemented, message transactions form one of the most efficient communication protocols for local-area networks, both in terms of delay and of throughput [6, 2]. In many popular implementations, when a client process has sent a request, it blocks until a reply arrives; when a server has asked for a request, it blocks until one arrives.

Using message transactions has several consequences for the design of the programming environment. First, processes block once on each message transaction. Two process switches thus occur: one when the process blocks to run another process and one after the process has become unblocked again to run the original again. If message transactions are to be very fast, process switching had better be fast too.

Second, message transactions provide no parallelism: when the client runs, the server waits for a request, and when the server runs, the client waits for a reply. Only one process runs at a time, albeit on different machines. One solution could be to implement non-blocking transactions, thus killing two birds with one stone: process switches need not compete with message transactions in speed any more and parallelism can be obtained by sending requests to many servers simultaneously. This solution, however, introduces a whole new set of problems [13]. One problem is that the interface between a process and the communications substrate becomes more complicated: there must be handles for telling a process when a message has arrived. Another is that the number of process switches does not decrease at all: the communications software (which must reside in a separate address space

† Unix is a Trademark of AT&T Bell Laboratories.

or in the kernel for protection) is invoked upon requests to send, requests to receive, and upon receipt of a message from the network. A third problem is that a non-blocking message transaction interface is extremely hard to program and debug, because the order of events is no longer specified.

Parallelism must be provided in some other way, and the way that was chosen in many modern distributed systems is to implement *light-weight processes*. Many light-weight processes can share a single address space; since much of the state of the light-weight processes is shared, switching between light-weight processes can be done blindingly fast. Using light-weight processes makes it possible to implement servers by having one process serve a single client at a time; many clients can be served simultaneously because there are many parallel light-weight processes. Usually, a synchronization mechanism is provided to allow the processes to share common data structures in shared memory (*e.g.*, in the form of *semaphores*).

Light-weight processes and blocking message transactions are used in many distributed systems to simplify writing software that exploits parallelism [12, 14, 13].

Mechanisms for migration of processes in distributed systems have been proposed or implemented several times [0, 10, 11, 1], but no algorithms have been proposed to use migration for load-balancing. Given the time required to migrate a large process (on the order of ten seconds), migration for load-balancing does not appear to be very useful. It can be useful, however, in an environment of personal workstations, where idle workstations are 'lent out' as a processing resource for others and 'taken back' when their owners return.

3. The Amoeba Distributed Operating System

Amoeba is a distributed operating system, based on the popular paradigm of *client* processes communicating with *services* via message transactions. Amoeba uses *capabilities* to access services and the objects these services implement.

A *capability* is a 128-bit reference to an object; the first 48 bits – known as the *port* – refer to the service managing the object; the remaining 80 bits are allocated by the service to identify the object. Both parts of a capability are generated in such a way – and contain sufficient bits – that the probability of an unauthorized user guessing an object's capability is negligible.

These capabilities are used for protection, and also as the primary mechanism for addressing requests to do operations on objects. When a client sends a request, the system uses the port to determine which service should handle the request. A server for that service is then found through a *locate* operation, *e.g.*, implemented through broadcasting *'where-are-you'* packets. The server uses the private part of the capability to identify the object. After carrying out a request, the server returns a reply.

Most services run in user space. The Amoeba Kernel provides only the bare minimum of service: message-transaction facilities, process management, and access paths to peripherals. File service, for instance, is a user-space service with no special privileges, except knowledge of the capabilities to get to the disks where the files are stored.

Message transactions are blocking, and the system provides no buffering. When a server calls *getrequest*(*port*, *capability*, *requestbuffer*), (the port identifies the server to the system), the server is blocked until a request arrives. The server returns a reply with *putreply*(*replybuffer*), which doesn't block. When the client calls *trans*(*capability*, *requestbuffer*, *replybuffer*), it blocks until the server's reply is received.

In case of a failure, the client is told that the server could not be reached, or that no reply was received. In the former case, the client can safely retry; in the latter case, the client will have to find out whether the failure occurred before, during, or after execution of the request (unless the request was *idempotent*; in this case the request can always be safely repeated). When a client fails during a transaction, the reply is lost.

A kernel request is just a request for an operation on an object maintained by the kernel. A kernel request, or *system call*, is a transaction with the *Kernel Service*. Thus, Amoeba only has the system calls for doing message transactions.* This allows us to implement services as kernel-space services or

* Well, not quite. For efficiency, a few calls have been implemented as system calls; see § 3.3, for instance.

user-space services without changing any user code. It also allows us to use the same capability-based protection for system calls and other calls.

Since Amoeba transactions† are blocking, they cannot be used to obtain parallelism. Amoeba uses parallel processes to achieve that. Amoeba implements light-weight parallel processes, called *tasks*. For efficiency, a number of tasks can share an address space. An address space with a number of tasks in it is a *cluster*. Because the term *process* could refer both to a task or a cluster, we have avoided it as much as possible in the remainder of the paper.

To allow programmers to use separate tasks for small units of work (*e.g.*, use a separate task for each request received by a file server), tasks are cheap to create, destroy and schedule. The current scheme for this is quite efficient, but we believe it can be made more flexible and more efficient still. This paper discusses a new design for task and cluster management.

For more information about Amoeba, see 'The Design of a Capability-Based Distributed Operating System' [9]. For details of the Amoeba protection mechanism, see 'Protection and Resource Control in Distributed Systems' [7].

4. The Kernel Server

The Amoeba Kernel manipulates three kinds of basic objects to realize the process abstraction in Amoeba. A *cluster* is a virtual address space consisting of a number of *segments* and a number of threads of control, called *tasks*.

The reason for having tasks share an address space is one of efficiency: Tasks can exchange information among each other more efficiently in shared memory, and, since tasks have little context, task switching can be made faster. The concept of tasks is used in several modern distributed systems, notably, V [14], Mesa [0], and Topaz.*

4.1. Segments

A *segment* is a named linear section of memory. It is an object, managed by Kernel Service. When the kernel server receives a request to create a segment of a certain length, it allocates that amount of (virtual) memory, possibly rounded up to a whole number of pages, and hands the requestor a *capability* for it. The capability proves ownership of the segment and allows the holder to manipulate it.

Segment Capability
Virtual Address
Length
How Mapped

Figure 1. Segment Descriptor

After a segment has been created, it can be mapped into a cluster's address space, using a *map request*. The map request includes a data structure that consists of the segment capability and its starting address and length. It also tells *how* the segment is to be mapped (*e.g.*, read-only, read-write, trap-on-write). This data structure is called a *segment descriptor* and is depicted schematically in Figure 1.

Segments can be unmapped again, and they can be deleted. The actual removal does not take place, however, until the segment is no longer mapped into any cluster's address space. Segments cannot migrate; they are created and deleted on the same (multi)processor. In § 5.1, we show how we can achieve a useful sort of 'virtual migration' of segments.

† Not to be confused with database transactions or atomic transactions. In Amoeba, a transaction is a message transaction, a request/reply pair.

* Unfortunately, no papers on this very interesting multiprocessor operating system have been published to date; the developers at DEC's System Research Center need every possible encouragement to remedy this.

4.2. *Clusters*

The Kernel Service also manages clusters, which are created by sending a *CreateCluster* request to the kernel server. The parameter to the request is a *cluster descriptor* which describes the initial state of the cluster by describing the state of its tasks, the address space in which these tasks will run, and the processor on which the cluster must run. Figure 2 illustrates a cluster descriptor.

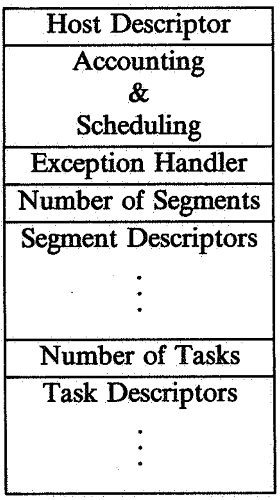

Figure 2. Cluster Descriptor

The *host descriptor* describes the kind of processor the cluster runs on. Its entries have a type and a value. The type *instruction set*, for instance, assumes values such as *VAX*, *M68000*, or *NS32000*, and describes the instruction set to which the cluster's code belongs. An instruction-set-dependent *options* type is used to indicate whether the cluster will need instruction-set options like floating point or extended instruction sets. The *memory size* type has a value that indicates the maximum size that the cluster's address space may need to grow to. There are many more possible types; new types can easily be added. The Kernel Service recognizes a number of useful types and uses their values to determine whether it can or will handle the cluster. Other types may be used by user services that manipulate cluster descriptors (*e.g.*, Process Service).

The *accounting & scheduling* field contains information about just that. It is not really used at the moment, but we envisage that one of its uses can be to provide scheduling information from a previous host to the next one when a cluster migrates. Another use is as a measuring device for execution times of clusters.

The *exception handler* field gives the port of the service to handle exceptions if they occur.

Then follow the *segment descriptors*, one for each segment in the cluster's address space. When a segment is handed to a kernel server, the segment capabilities must refer to local segments. As shown in § 3.3, the segment descriptor tells where the segment goes in the cluster's virtual address space, how long it is, and how it must be mapped.

Finally, a list of *task descriptors*, one for each task in the cluster, gives the state of each task in the cluster. This illustrates one of the advantages of having transactions as the only way to communicate outside a cluster: the Amoeba Kernel maintains very little state for the tasks. The state consists only of whether it is runnable or blocked on a semaphore or condition variable, the value of the program counter, the stack pointer, processor status word, the other registers and, if a transaction is in progress, its state. Note that a task can be involved in only two transactions at a time: It can be doing a transaction with a server while serving a request for a client itself. State has to be maintained for both ongoing transactions. Later, we shall return to the issues of starting and stopping clusters with tasks that are in the middle of a transaction.

When a *CreateCluster* command is given, the cluster descriptor's segment capabilities must refer to existing segments. Thus, the kernel server has all the information it needs to start up the new cluster. It returns a *cluster capability* to the client issuing the request, so that only this client, as the owner of the new cluster, can exert control over the cluster.

4.3. The Kernel Server Interface

Transactions with Kernel Service

Cluster creation and deletion (cluster may delete self):
CreateCluster(*KernelCap*, *ClusterDesc*); returns *ClusterCap*
DeleteCluster(*ClusterCap*); returns *ClusterDesc*

Segment management:
MapSegment(*ClusterCap*, *SegmentDesc*, . . . , 0); returns *ack*
UnMapSegment(*ClusterCap*, *VirtualAddr*, . . . , 0); returns *ack*
DeleteSegment(*SegmentCap*); returns *ack*

Interrupting clusters:
Signal(*ClusterCap*, *SignalType*, *Parameter*); returns *ack*

Kernel System Calls

Task management:
MakeTask(*Program Counter*, *StackPointer*); returns *ack*
ExitTask(); does not return

Synchronization:
P(*Semaphore*);
V(*Semaphore*);
Sleep(*Condition*);
Wakeup(*Condition*);

TABLE 1. Kernel requests and system calls for process management.

TABLE 1 lists the interface with the kernel server for process management. The first argument to a request is the capability of the object the request refers to. The *CreateCluster* request refers to an object that does not yet exist; its capability is a Kernel Capability that provides protection against unauthorized clients spawning clusters on kernels they have no access to. *Ack* indicates a generic success or failure reply. In case of failure it gives a reason as well.

Half the system calls are implemented as transactions with the kernel, the other half as traps into the kernel. The reason for implementing some of the calls as traps is one of efficiency. Actions such as *MakeTask* or *P* are executed very very often and need to be implemented in the absolute minimum number of instructions possible. Note that the system calls have effect only withing the cluster that issues them. The transaction calls need the protection of the Amoeba protection mechanism.

CreateCluster creates a new cluster. Its argument is a cluster descriptor that describes the cluster to be started. For each task of the cluster, the descriptor includes a task descriptor giving program counter, stack pointer and register contents are given, and for each segment, the mapping information is present in the segment descriptors. The Segment Capabilities must refer to locally existing memory segments. A null-capability is also allowed and indicates a request to create a zero-filled segment along with the cluster. In § 5.2 we shall return to the problems of creating clusters in an arbitrary state, such as needed for migration.

DeleteCluster deletes a cluster and returns its cluster descriptor. The cluster descriptor returned could be given to a *CreateCluster* command and the cluster would continue where it was stopped, if it weren't for the fact that other clusters communicating with this one may have been told that it was killed between the *DeleteCluster* and the *CreateCluster.* Suspending or migrating a cluster is trickier than this. The details are described in § 5.2.

Segments are created by a *MapSegment* command with a null capability. Existing segments are mapped by naming their capability in the appropriate segment descriptor. The segment descriptor further specifies the virtual address of which segment should be mapped and the length of the mapped memory segment. If the map length is more than the actual size of the segment, the segment is grown and the added piece is zero-filled. On some machines, stacks run from high addresses to low and have to be grown at the bottom end; segments can be mapped read-only, trap on access, etc. The *how* field specifies how the segment should be mapped. A mapping can be changed by mapping the same segment at the same virtual address with different *how* or *length* fields.

UnMapSegment removes a segment from an address space again. *DeleteSegment* throws a segment away. Segments are not actually deleted until they are no longer mapped into any address space.

The execution of a cluster can be interrupted by sending a signal. A signal causes a cluster to freeze in its tracks and its state to be sent to a *debugger server.* To handle the signal, the debugger can inspect and change the state of the cluster before allowing it to continue execution. The signal- and exception-handling mechanism is described in § 4.

The Kernel Service transactions described above are protected by the normal capability-based protection mechanisms of the Amoeba system: An application can only create clusters on those processors for which it has a Kernel Capability. An unauthorized user can thus easily be prevented from running clusters on another user's workstation, for instance. Segments are also protected with the capability mechanism. One user's private segment can not be mapped by another without explicit permission. Signals can only be sent to clusters by holders of an *owner* capability for the cluster.

The calls we are about to describe do not need this heavy-weight protection mechanism, because they only affect the cluster from which they are done. The task management calls and task synchronization calls could therefore be safely implemented as 'real' system calls, which is fortunate, because their efficient implementation is critical to the performance of Amoeba.

A new task is created with a *MakeTask* system call. The parameters are a program counter and a stack pointer. The new task will start execution at the address indicated by the program counter. A new task cannot be started in the middle of a transaction; registers are undefined. A task can delete itself by an *ExitTask* call.

For synchronization, four calls are provided: *P* and *V*, operating on binary semaphores, and *Sleep* and *Wakeup* on condition variables [4, 5]. *Sleep* puts a task to sleep and *Wakeup* wakes up every task sleeping on the condition. These primitives are essentially the same as those in the Topaz distributed system, and its predecessor, Mesa [0]. In the normal case (no contention for the semaphore), *P* and *V* execute completely in user space. A system call on *P* is only necessary if the semaphore has already been acquired by another task; on *V*, one is necessary only if another task is blocked waiting for it. We stole the idea for this optimization from Topaz.

5. The Debug Server

When an Amoeba cluster traps because of an exception, a debugger is automatically invoked. The Debug Server, a user-space cluster with no special privileges, can reside on the same kernel as the faulty cluster, but it can also be remote. For remote debugging, however, some help from the Process Server is desirable. In this section, we shall describe the mechanisms for handling exceptions and signals.

Exceptions and signals are different, but handled identically. An *exception* is essentially a synchronous event, caused by a cluster to itself. Typical exceptions are division by zero, addressing nonexistent virtual memory, attempting to execute non-instructions, etc. A *signal* is an asynchronous event, caused by a source external to a cluster. Signals are typically caused by humans hitting the *interrupt* key on their terminal and they are meant to terminate execution of a cluster, or at least make it interrupt its normal flow of execution. Signals play an important role in migration, as we shall see in the next section.

Signals and exceptions interrupt the execution of a cluster. Exceptions generally cause a hardware trap, which is handled by the kernel. Similarly, signals also end up in the kernel on which the cluster executes. Both signals and exceptions cause the following things to happen:

1. All running tasks in the cluster stop execution. On a multiprocessor, it is not possible to stop all tasks atomicly; here, we attempt to stop the tasks as quickly as possible.
2. Active transactions are *frozen*: the transaction protocol replies to incoming messages with a *'try again later, this cluster is frozen'* response. This will cause the sending protocol entities to retry sending the same message later, repeatedly, without giving up as long as this reply is given.
3. A cluster descriptor for the signaled cluster is made and the Kernel sends a *PleaseDebug* request to the server whose capability was in the *signal capability* field of the cluster descriptor when the cluster was created.
4. The Kernel then waits for a reply from the Debug Server, which may contain a modified cluster descriptor. After incorporating the modifications in the state of the cluster,
5. The cluster resumes execution, possibly in a modified state.

Ongoing transactions are only frozen in a few well-defined states: Servers can be frozen while waiting for an incoming request (but not after the request has started coming in), or while processing a request (between the completion of *getrequest* and the call of *putreply*). Clients can only be frozen between when sending the request has completed and the reply starts coming in. Further, clients or servers cannot be frozen while the protocol is waiting for an acknowledgement. Clusters that are neither client nor server (*i.e.*, in between transactions) can always be frozen. Note that transactions thus cannot be frozen if messages may have to be retransmitted (waiting for an acknowledgement). Note also that the times during which transactions may not be frozen are bounded in length (by maximum number of retransmissions, maximum number of packets in a message, retransmission time and maximum packet life time) and are generally short.

The replies the Debug Server can give to the *PleaseDebug* request are *continue* or *delete*. The former allows the cluster to continue execution; if a modified cluster descriptor accompanies the reply, the state of the cluster is first adapted. The latter does not restart the cluster but deletes it.

6. The Process Server

Every Amoeba machine has an instance of the Process Server running on it. The Process Servers implement the lowest level of process management policies in Amoeba: Process Service may or may not support cluster migration, cluster checkpointing, etc. However, the Process Server always provides a few basic and necessary facilities; we shall discuss these first.

As shown in the previous section, a cluster is created by creating and writing a number of segments, followed by a request to make a cluster containing those segments. There is no Kernel mechanism to write into a remote segment. Cluster creation, therefore, can only be done with the help of a local agent. The Process Server is this agent, normally.

6.1. Global Segments and Kernel Segments

The Kernel Server implements local segments; the Process Server uses segments to implement a system-wide segment abstraction. To distinguish them, we shall refer to the Kernel Server segments as *local segments* and to the Process Server's system-wide segments as *global segments*.

The Process Servers maintain an important invariant for global segments: Two global segments have the same name only if they have the same contents. (The reverse need not hold: segments with different names *are* allowed to have the same contents.) Thus, when a cluster is created with a global data segment A in its address space, and subsequently modifies that segment while running, the global data segment it has while it migrates to a new host will have a new name, B, different from A.

The most important task of the Process Server is to cause clusters to be started on the machine they serve. Essentially, this is done by sending a *RunCluster* request to the Process Server. This request contains a cluster descriptor just like the one described in the previous section. The only difference between this cluster descriptor and the one that will later be given to the Amoeba Kernel is that this one contains global segment capabilities, while the one that will later be given to the kernel must contain local segment capabilities.

But this is easily remedied: The Process Server creates and initializes the required number of local segments, replaces the capabilities in the cluster descriptor it received and passes it then on to the Kernel Server in a *MakeCluster* request.

Process Servers initialize segments by mapping them in their own address space, writing into them and unmapping them again. When the segment contents are fetched from a remote server by doing a transaction, this is especially efficient, since the reply buffer with the data can be mapped directly onto the segment where the data should go. Thus, no in-core copying is needed.

Most clusters have read-only code segments. After running a cluster, unmodified code segments can be kept around in case the same program is run again on that kernel. This is the case for segments that were mapped read-only and for read-write segments that have not been written into. Paging hardware often provides a mechanism to inform the operating system whether or not a segment was modified (the dirty bit). The Amoeba Kernel provides the dirty bit in the segment descriptor when it passes a Cluster up to a user task. The Process Servers will not change a global segment capability if its contents have not changed. When a cluster thus migrates from machine to machine, each time the global code segment capabilities will remain the same. The global data and stack segment capabilities for the segments whose contents have changed will be new and unique for each migration, even though the cluster itself thinks of it as one and the same segment all the time.

6.2. Migration

Although clusters rarely move to a new host after being started up, migration is a central concept in the Amoeba process management mechanisms. This is because loading new clusters into memory, taking core dumps, making checkpoints, and doing remote debugging are all similar to migrating a cluster. In fact, if we can migrate a cluster from one machine to another, downloading, checkpointing, debugging, etc., should be simple.

Load balancing by migrating cluster is a poorly understood area and it is dubious whether it is very useful with the current sort of workstations and networks. Migrating a five megabyte cluster, for instance, will take at least ten seconds, because that is how long it takes a fast transport protocol to copy the memory contents over a 10 Mbit Ethernet; five megabyte programs are not at all uncommon, especially as candidates for migration: long-lived clusters are usually large too. Migration is thus rather expensive, and the gain of a migrate operation must be big in order to merit one.

In spite of this, we believe that migration can be useful. When a workstation's owner logs off in the evening, the workstation can turn itself into a Pool Processor and provide process-execution service to the rest of the system. When the owner returns in the morning, however, and logs back on, the guest clusters running there could be nudged off by migrating them away to some other workstation [11].

The Process Servers implement the cluster migration mechanism. It does not implement a policy; the decision to migrate a cluster and where to migrate it is made in a higher level of service. We shall not go into how this decision is made.

When a cluster moves from one machine to another, the Process Server at the old machine makes the memory contents and cluster descriptor of the cluster available to the Process Server on the new machine. The Process Server on the new machine loads the cluster into memory and starts it off. We will call these Process Servers *oldPS* and *newPS*. We will examine the migration cluster from the point where *oldPS* has received a request to migrate the cluster to *newPS*. *OldPS* has to set things up to handle the cluster's signals as the cluster's debugger.

First, *oldPS* sends a signal to the cluster, which causes the Amoeba Kernel to freeze it in its tracks and send a cluster descriptor to *oldPS* (*oldPS* acts as the debugger for this cluster). Then, *oldPS* creates global segment capabilities for the cluster's segments, keeping in mind that segments whose contents (may) have changed must be given new, unique capabilities, while segments whose contents have not changed may retain their previous global capability. Every Process Server has an internal Segment Server task which serves requests from remote clients to obtain the contents of segments. When the contents of a particular segment are needed, the Segment Server maps them into its virtual memory and replies to these requests with the reply buffer in the mapped segment. Finally, *oldPS* sends the cluster descriptor — the local segment capabilities replaced by the global segment capabilities — to *newPS* in a *RunCluster* request.

NewPS, when it receives the *RunCluster* request, examines the global segment capabilities to see if it already has any of them around. If so, it uses those; otherwise, it creates them and maps them into its address space. It then sends *ReadSegment* requests to the Segment Server (a task in *oldPS*), with the reply buffer in the mapped segment. With both client and server sending and receiving directly out of mapped memory, a cluster's memory contents can thus be copied over our Ethernet at speeds well above half a megabyte per second.

When all the segment contents have been copied, *newPS* replaces the global segment capabilities in the cluster descriptor by the appropriate local one and starts the cluster with a *CreateCluster* request. When this is done, migration is complete and the cluster executes on its new host. Finally, *newPS* returns a reply to *oldPS*. *OldPS*, when it receives its reply, can delete the cluster with a *DeleteCluster* request to the Amoeba Kernel.

Note that, while migration was in progress, the cluster existed on its old host in 'frozen' condition. The kernel thus replied to all messages for the frozen cluster with a *'try again later, this cluster is frozen'* message. Now that the cluster has been deleted, those messages will come in again at some point, and the kernel will now reply with something like *'this port is unknown at this address.'* The sender will then do a *locate* operation to find the new whereabouts of the cluster, and communication will be re-established.

The protocol for dealing with message transactions during migration is more subtle than described here, but would take too much space to describe fully. To preserve the *at-most-once* semantics of Amoeba message transactions, client and server need to use unique communication ports so that the locate operation cannot yield the address of the wrong server, for instance.

6.3. The Program Server

Program Service provides the system with new clusters. It stores the binaries of the programs running in Amoeba on files and makes them available to Process Servers in such a way that the Program Server appears to them as a Process Server. Downloading a program over the network uses exactly the same mechanisms as migration.

In a heterogeneous Amoeba with different kinds of machines, the Program Server can keep several versions of a program around, one for each type of machine. When deciding where a program should run, the Program Server can be queried about the versions it has for the program.

Clusters can be suspended and given to the Program Server for temporary storage. They can continue their execution later, when they are downloaded and run like any other program. This is not only useful when making checkpoints, it can also be used to allow a program to do some complicated initializations before it is stored by the Program Server. A program, such as *troff*, for instance, which reads a complicated macro file before typesetting a document, can be speeded up considerably by storing it in the state where it has just finished reading the macro file.

Note that there is little point in storing a cluster with tasks waiting for a reply: when it is continued a week later, it is extremely unlikely that the server that is supposed to produce that reply has waited patiently all that time to deliver the reply. However, it is perfectly proper to do this with tasks waiting for a request to come in, since there is no process that has to keep transaction state while the cluster is in cold storage.

7. Emulation Service

One of the most important applications of the rather general mechanisms for handling signals, traps and exceptions in the previous section is that it allows the emulation of any operating system environment. Amoeba was developed in a Unix environment, which is why we have concentrated on Unix emulation, but there is no reason why any other operating system interface could not be emulated.

We have implemented two forms of Unix emulation: by intercepting the system calls at the level of the C source code, or at the level of the system call. The former is simpler to realize and – combined with tailored supporting services – gives adequate performance. The latter is more complicated, but it can be used to provide *binary compatibility*: binaries that run under ordinary Unix can be made to run under Amoeba without changing a single bit.

We have done both under a previous version of the Amoeba Kernel. The library for Unix emulation at the source-code level will remain practically unchanged under the new process management dominion. The Kernel version that Unix emulation runs on now maintains a table of {*task capability, emulator capability*} pairs. When a task traps, and an entry is found in the table, the registers (PC, SP, PSW and general purpose registers) and the address of the interrupt vector are sent to the emulator. The emulator uses transactions with the *Segment Server* (a server for reading and writing memory which will be replaced by the Process Server under the new process management regime) to get at the memory contents of the cluster. It returns new values for the registers to the kernel. The emulator itself runs on Unix, which was modified to allow doing transactions. The emulator interprets the system calls given to it by doing them on Unix and passing the results back. One of the simplest system calls in Unix, *getpid()* takes about 10 ms when emulated this way, most of which is taken up by the transaction protocol code in the Unix kernel.

Both in this scheme and the new one, the Amoeba Kernel has no knowledge whatsoever of Unix system calls. It merely invokes the debugger when a user task traps. The differences between the working system and the one we are implementing are the following: In the old one, processes to be emulated are created through the emulator which keeps track of most of its state; the state given to the emulator consists of just the registers. In the new scheme, the state will be the cluster descriptor. Clusters to be emulated need not be created by the emulator. In the old scheme, memory is read through transactions with the Segment Server. In the new one, memory can be read and written directly by the emulator, because it is mapped into its own address space.

When we have some experience with this arrangement, we will decide if this new path through the Kernel to the Unix emulator is too long. If so, we shall have to construct a representation of a *lightweight state* that can be given to the emulator instead of the current, rather heavy-weight, cluster descriptor. In any case, the emulator will have the emulated cluster's memory mapped into its own address space as well, providing very efficient memory access.

8. Conclusions

This paper reveals the tip of an iceberg. Building a coherent set of primitives for process management that includes migration of clusters, checkpointing, debugging and emulation of arbitrary operating system interfaces involves very careful design, not only of the mechanisms that deal with process management directly, but also with all of the surrounding environment. In this section, we shall attempt to lift out some of the design considerations that made it possible for us to design the system as we did.

The Amoeba Kernel provides a minimum of functions: process management and interprocess communication. There is thus also a minimum amount of state that has to migrate when a cluster migrates. We believe that this was one of the essential choices that made our mechanisms work. Things would have been much more difficult if we had to deal with things like 'open file state,' 'controlling terminals' or the complicated connection state of a sliding-window protocol.

The Amoeba interprocess communication mechanism has also been vital to the success of our design. First, the communicating entities are named using a location-independent naming mechanism that uses an underlying *locate* service to find out dynamically where the packets have to be sent. None of the migration apparatus has to worry about rerouting messages, no forwarding addresses have to be left behind [10]; ex-hosts can forget about the existence of a cluster immediately after migration is complete.

Second, the simplicity of the Amoeba protocols contribute enormously to the portability of clusters. The protocol has only a few states in which it can stay for arbitrary lengths of time and it is relatively easy to migrate a cluster in these states using the *'I'm frozen, don't bother me'* messages described earlier. When the protocol is in any of the other states, the Amoeba Kernel can wait until the protocol reaches a 'migratable' one.

The most important conclusion we have drawn from this design – which is still being implemented – is that it is possible to build a simple mechanism that is sufficient to realize downloading, migration, exception handling, checkpointing, emulation and debugging. Although the implementation is not complete at the time of writing this paper, we expect to finish soon enough to present performance information at the SOSP conference.

9. Acknowledgements

Jack Jansen (who is implementing the system discussed in this paper), Robbert van Renesse and I have had many discussions about the Amoeba process management design which improved it considerably.

Dave Redell and Lucille Glassman read the draft of this paper and suggested numerous changes to make it readable.

References

1. D. A. Butterfield and G. J. Popek, (June 1984). Network Tasking in the Locus Distributed UNIX System, *Proc. Summer USENIX Conf.*, 62-71.
2. David Cheriton (January 1987). VMTP: Versatile Message Transaction Protocol, Preliminary Version 0.3, *Computer Science Report*, Stanford University.
3. D. R. Cheriton and W. Zwaenepoel, (October 1983). The Distributed V Kernel and its Performance for Diskless Workstations, *Operating Systems Review*, 17.5, 129-140.
4. E. W. Dijkstra, (1972). Hierarchical Ordering of Sequential Processes, in *Operating Systems Techniques*, Academic Press, New York.
5. C. A. R. Hoare, (August 1978). Communicating Sequential Processes, *Comm. ACM*, 21.8, 666-677.
6. S. J. Mullender and R. van Renesse, (1984). A Secure High-Speed Transaction Protocol, *Proceedings of the Cambridge EUUG Conference.*
7. S. J. Mullender and A. S. Tanenbaum, (1984). Protection and Resource Control in Distributed Operating Systems, *Computer Networks*, 8.5,6, 421-432.
8. S. J. Mullender, (October 1985). *Principles of Distributed Operating System Design*: Stichting Mathematisch Centrum, Amsterdam.
9. S. J. Mullender and A. S. Tanenbaum, (1986). The Design of a Capability-Based Distributed Operating System, *The Computer Journal*, 29.4, 289-300.
10. M. L. Powell and B. P. Miller, (October 1983). Process Migration in DEMOS/MP, *Proc. 9th ACM Symp. on Oper. Syst. Prin., Oper. Syst. Review*, 17.5, 110-119.
11. M. M. Theimer, K. A. Lantz, and D. R. Cheriton, (December 1985). Preemptable Remote Execution Facilities for the V-System, *Proc. 10th ACM Symp. on Oper. Syst. Prin., Oper. Syst. Review*, 19.5, 2-12.
12. R. W. Watson and J. G. Fletcher, (February 1980). An architecture for Support of Network Operating System Services, *Computer Networks*, 4.1, 33-49.
13. R. F. Rashid and G. G. Robertson, (December 1981). Accent: A Communication Oriented Network Operating System Kernel, *Proc. 8th ACM Symp. on Oper. Syst. Prin., Oper. Syst. Review*, 15.5, 64-75.
14. B. W. Lampson and D. D. Redell, (February 1980). Experience with Processes and Monitors in Mesa, *Comm. ACM*, 23.2, 105-117.