Experiences with the Implementation of a Process Migration Mechanism for Amoeba

Chris Steketee, Piotr Socko† and Bartosz Kiepuszewski†

Advanced Computing Research Centre,
School of Computer and Information Science,
University of South Australia, The Levels SA 5095

Chris.Steketee@cis.unisa.edu.au

† Visiting from Institute of Informatics, Department of Mathematics, Informatics and Mechanics, Warsaw University, Poland. {sociek, bartek}@mimuw.edu.pl

Abstract

We describe our experiences with the implementation of a process migration mechanism for the distributed operating system Amoeba. After describing our design goals, we present our implementation for Amoeba. Though our goals have very largely been met, we have fallen short of the goal of complete transparency, and we discuss the consequences of that. We also present performance figures, indicating that the speed of process migration is limited only by the throughput of the network adapters used in our configuration, and that the overhead is comparable to that of process creation. We conclude with a review of the degree to which our design goals have been met, and discussion of the lessons learnt.

Keywords process migration, distributed operating systems, performance.

1 Introduction

Process migration is the movement of an executing process from one host processor in a distributed computing system to another. This paper describes our experiences in implementing process migration for the distributed operating system Amoeba.

Our main interest in process migration is to assess its applicability to load balancing, to allow experimental assessment of the performance of various load balancing algorithms.

In previous papers, we presented the design of a process migration mechanism for Amoeba, giving the results of a prototype implementation [1], and reported the results of preliminary load balancing studies using the prototype [2, 3]. The conclusions of these studies, like earlier results [4-6], were equivocal about the usefulness and applicability of process migration to load balancing. An important factor (not surprisingly) is the performance of the process migration mechanism - the elapsed time to migrate a process, and the amount of overhead imposed.

Since then, we have completed a full implementation of process migration based on the same design, and have begun further load balancing studies using the new implementation. The new implementation differs from the prototype in two important respects - it has much better performance, and it deals properly with the migration of processes engaged in communication. This paper concentrates on this implementation, presents performance results, and has some comments to make about the lessons learned. A future paper will present the new load balancing studies.

2 Overview of process migration

The term *process migration* means the movement of an executing process from one host processor (the *source*) to another (the *destination*), followed by its continued execution on the destination. It is to be distinguished from *process placement*, which is the selection of a host for a new process and the creation of the process on that host. Both facilities may be used for load balancing, but it is the former which is the subject of this paper.

Process migration has been the subject of a considerable amount of research, and there have been a number of implementations reported in the literature [7-18]. Applications of process migration include dynamic load balancing (spreading load more evenly over a set of hosts), improved reliability (allowing long running processes to survive the planned shutdown or impending failure of a host), and improved access performance (for example moving a small process to the site of a large data file) [19-21].

Process migration is considerably more difficult to implement than process placement, since it involves a process in a state of execution. Migrating a process requires suspending the process on the source host, extracting its state, transmitting the state to the destination, reconstructing the state on the destination, resuming the process's execution on the destination, and deleting the process on the source. This sequence of events should be controlled in such a way that if it does not complete successfully, the process can continue its execution safely on the source host. All this should be done while keeping the overhead of process migration comparable with the overhead of creating and starting a new process. Finally, there is the need to correctly route messages sent to the process during and after migration.

A more detailed overview of process migration design issues may be found in [20], an annotated bibliography in [22], and a survey of systems providing process or object migration in [21].

3 Overview of Amoeba

Amoeba is a distributed operating system which has been under development at the Vrije Universiteit, Amsterdam since 1981. The current version, Amoeba 5, runs on Intel 80x86, Motorola 680x0, and SPARC platforms. In addition to research into distributed operating systems, Amoeba has been used for the implementation of parallel algorithms and as a platform for a distributed programming language, Orca [23, 24]. An excellent exposition of Amoeba can be found in [25]. In this section, we briefly summarise Amoeba concepts to give the background needed for the remainder of the paper.

3.1 Amoeba concepts

Location transparency: Amoeba is a distributed operating system as defined in [26]. It provides location transparency - neither end-users, nor application programs, need to know the location of the objects (including processes) being addressed.

Microkernel design: The Amoeba kernel contains relatively little functionality, with most traditional operating system functions carried out in server processes. The kernel performs low-level memory management, process scheduling, inter-process communication, and input/output device handling.

Objects, servers and capabilities: Amoeba is an *object-based* operating system. Objects are managed by *servers*, one for each type of object. In order to perform an operation on an object, it is necessary to have a *capability* for that object, identifying both the server managing the object (by means of a *port* included in the capability) and the object itself. Amoeba has a number of standard servers, including the Bullet file server [27], the Soap directory server [28], a process server and a time-of-day server. Servers may also be written by users when developing new applications.

Remote procedure call: Amoeba's basic mechanism for inter-process communication is the *Remote Procedure Call* or *RPC* [29]. This is implemented using synchronous message passing - a *client* process sends a *request* message to a server, which carries out the request and responds with a *reply* message. In Amoeba, this exchange of messages is known as a *transaction*.

Amoeba RPC is based on three system calls. The client process uses *trans* to send a request and wait for the reply. The server uses *getreq* to indicate its readiness to receive a request on a specific port, and *putrep* to send the reply message. All three calls are blocking. Amoeba uses "at-most-once" semantics - if a client does not receive a reply, it does not know whether the request has been carried out, but it is assured that the request has not been carried out more than once.

Fast local internet protocol (FLIP): The RPC mechanism is layered on top of a network protocol known as FLIP [30]. This provides processes with host-independent network addresses, which can migrate with processes. The RPC layer in Amoeba 5 maps ports to FLIP addresses rather than to hosts. The port is a permanent address for a service, whereas the FLIP address is associated with a particular incarnation of a process. The FLIP layer uses a broadcast mechanism to bind an address to a host, and caches the resulting mapping.

FLIP also provides multicast primitives, which are used in Amoeba 5 to implement an atomic group communication facility.

Processes and threads: The unit of execution is the process. A process resides completely on one host processor; several processes may co-exist on one host. A process occupies one or more memory segments which together constitute its address space. All segments for a process reside in physical memory for the entire period of time that the process exists - virtual memory is not provided. Memory management hardware is however used for protection and relocation.

A process may have several *threads* of execution within its address space. Synchronisation between threads uses *mutexes*, with defined operations *lock* and *unlock*. Multithreading is used to provide the parallelism that is otherwise lost by virtue of the synchronous nature of the RPC mechanism.

Process server: The functions of process management are performed by a *process server* which runs in each host. The process server runs in kernel mode because of its need to carry out privileged functions. In order to perform process-related functions such as process creation, clients perform RPC transactions with the appropriate process server.

3.2 Amoeba as a platform for process migration

Amoeba was not originally designed and implemented with process migration in mind. We have therefore had to retrofit process migration to an existing operating system. While this has caused some difficulties, Amoeba has nonetheless proved to be a suitable platform for implementing process migration. The reasons for this are:

• The microkernel design means that the kernel keeps relatively little process state; in particular, all file and device access uses RPC or group transactions to servers, so that the state of a process's input / output is completely described by its communication state;

• The fact that FLIP uses host-independent network addresses;

• The existence of a process checkpoint mechanism (see section 5.1) which allows the kernel state and memory contents of a process to be extracted and sent across the network.

The last two features were introduced in the current release, Amoeba 5, with process migration in mind, though this was not implemented.

4 Design Goals

This section presents the goals of our design. More detail on the design is to be found in [1].

Separation of policy and mechanism: We separate process migration *policy* from process migration *mechanism*. The mechanism is concerned with *how* migration is carried out, the policy is concerned with *when* and *where* to migrate *which* process. Separating them allows implementation of, and experimentation with, a range of process migration policies using one general mechanism. Moreover, it allows the policies to be implemented completely in user-level processes, whereas implementation of the mechanism involves modifications to the operating system kernel. Our interface between policy and mechanism is straightforward, requiring a policy to specify *P* - the process to be migrated, *S* - the source host, and *D* - the destination host.

User-level implementation: It is desirable to implement process migration as much as possible in user-level processes, noting however that implementation of the mechanism involves kernel modifications.

Location transparency: In a distributed operating system, users, and user processes, should not be concerned with where processes run - the operating system presents the abstraction of a single unified system [26]. Nor therefore should they be concerned with the occurrence of process migration. Our design goal is complete transparency - neither the process being migrated, nor processes with which it is communicating, should be aware of the occurrence of migration; no special programming should be required, and no programming restrictions imposed. Existing programs should not have to be recompiled or relinked in order to take part in process migration.

Residual dependencies: A residual dependency occurs when the migrated process continues to have some dependency on the host from which it migrated. For example, this may be used for redirection from source to destination host of messages intended for the process. Residual dependencies are undesirable for reasons of performance and fault-tolerance. Our goal is to leave no residual dependencies.

Performance: The implementation of process migration should be achieved with maximum possible efficiency. The ideal would be to achieve an overhead comparable to that of low-level process scheduling, but this is unattainable using 10 Mbps Ethernet. A more modest and realistic goal is for the overhead to be comparable with that of process creation on the destination host.

Homogeneous versus heterogeneous migration: In *homogeneous* migration, processes are migrated between processors of the same architecture, whereas *heterogeneous* migration allows processes to be migrated from one architecture to another. We restrict ourselves to homogeneous migration. While there has been some work on heterogeneous migration, for example [31], this has of necessity been restricted in scope, since it is clearly difficult to translate the execution state of an arbitrary process from one machine architecture to another.

5 Implementation

Migrating a process requires in essence (a) transfer of the complete state of the process from source host to destination host; (b) ensuring that messages for the process are directed to the destination host.

5.1 Transfer of State

The complete state of an Amoeba process consists of user state plus kernel state. The user state of a process is described completely by the contents of its memory segments plus the registers for each thread, and can be migrated by ensuring that its (virtual) memory addresses are the same on the destination host as they were on the source host. Kernel state includes the state of the process's communication with other processes, and therefore its input / output.

Checkpoint / restart mechanism: Amoeba has an existing *checkpoint / restart* mechanism with much of the functionality required for process migration [32, 33]. Any process with a suitable capability for a target process to be checkpointed can send a *stun message* to that process. The result is that the process is suspended and a *checkpoint message* is sent to the target process's owner (eg a shell), containing the kernel state of the target process (register contents for each thread, capabilities for its memory segments, etc). The owner may then terminate the process or resume it. The segment capabilities give access to the memory of the target process by means of RPC transactions to its process server. This is typically used to produce a memory dump on disk, or (by a debugger) to read and/or alter memory contents before resuming execution.

The checkpoint message may also be presented to a low-level process creation primitive (*pro_exec*) to request creation of a new process on a host with the same architecture. *Pro_exec* is a RPC to the process server on the new host, which performs RPC's to the process server on the original host in order to read the original process's segments; these, plus the kernel state in the checkpoint message, define the initial state of the new process.

When used in this way, checkpoint in fact provides a rudimentary process migration mechanism. It is simple to write a migration server which stuns a process, collects the checkpoint, presents that to the destination host as a new process, and terminates the process on the source host.

Not surprisingly therefore, the checkpoint mechanism provides the basis of our process migration implementation. However the mechanism as it stands is incomplete. In the first place, the checkpoint message does not contain all of the kernel state - in particular, communication state is not included, so that the new process cannot complete RPC transactions in which the original process was engaged. Secondly, no provision is made for the fact that processes in communication with the migrating process may experience a hiatus in communication while migration is going on.

Migration checkpoint / restart: In order to overcome these shortcomings we implemented a migration checkpoint mechanism, which is a straightforward (in principle) variant of the normal mechanism. The main points to be noted are:

- The migration variant of stun extracts RPC communication state from the kernel on the source host and includes it in the checkpoint message;
- A new process restart primitive (*pro_migexec*) has been implemented: this is similar to the existing *pro_exec* for process execution, but also installs the communication state into the kernel on the destination host (see 5.2)
- A flag is set in the source kernel to indicate that the process is being migrated; this is used by the communication software to reject attempts to send a message to the process (see 5.2).

System call: A difficulty in encapsulating and migrating kernel state arises when a thread is in a system call in the kernel, either executing a system call or blocked (execution suspended) waiting for some event. In either case the kernel state includes kernel execution information such as return addresses and procedure parameters for kernel procedures. This information is difficult to migrate; in particular, kernel addresses are not in general the same on different hosts.

Fortunately, most system calls are of short duration and it is satisfactory to let them complete before checkpointing the process. The problem arises with the remaining system calls - those that may block a thread. The calls in this category perform communication and thread synchronisation functions. It is not satisfactory to wait until these complete, since the delay can be indefinitely long (it is legitimate for a server under Amoeba to take a very long time to reply to a request).

The designers of the checkpoint mechanism, faced with this difficulty, implemented two types of stun - a *normal* stun, which waits for system calls to complete, and an *emergency* stun, which aborts a system call if a thread is blocked in it. Unfortunately neither is ideal for our purposes - with normal stun migration could take indefinitely long, and with emergency stun the process may see an error response from a system call after migration.

The best solution would obviously be one which allows blocked system calls to continue properly after migration. This would require a redesign of the system call mechanism, so that the kernel state of a blocked thread could be encapsulated in a manner which can be migrated (for example, no kernel addresses). While this is clearly possible in principle, it is a task we were unwilling to attempt in the time available to us. We have therefore chosen to adopt emergency stun for process migration, accepting the loss of a degree of transparency.

The consequences of this decision are that a process may receive an error return from a blocking system call as a result, not of a genuine error but of process migration. In many cases, this will not matter very much, and a process programmed to deal with normal error conditions will not need to be changed at all to handle the additional error case.

The main potential problem is where a RPC *trans* call is aborted - the migrating process has no way of knowing whether or not the requested action has been carried out. If the action is an idempotent one (eg read from a specified position in a file), then it is safe to repeat it; programs would typically retry idempotent transactions several times at an error return. For a non-idempotent action however (eg append a record to a file), any recovery action is dependent on the application logic; the simplest action is to report an error and terminate.

Aborting the RPC *putrep* call (if a server is migrated when it is sending a reply) has a similar effect. The server after migration will not be able to repeat the call; its client will see a *trans* failure and will have to deal with it as above. However, the RPC protocols do guarantee that under these circumstances a reply will not be partially delivered, nor will it be delivered more than once.

In fact, the impact of this loss of transparency is less than might be supposed; robust applications need in any case to have a way of dealing with transient error conditions caused by network failure / congestion or server overload, for example by avoiding non-idempotent transactions. Process migration simply adds another cause of transient error.

Transfer of memory image: Most of the time required for process migration is spent copying the memory image of the process from source to destination, since this is limited by network speeds. Implementations of process migration have used various techniques in an attempt to reduce this cost. Perhaps the most effective potentially is *lazy copying* as implemented in Accent [11] and Sprite [7], in which pages of the process address space are moved to the destination host only when referenced. The advantages and disadvantages of this and other techniques are discussed briefly in [1], and in more detail in [20].

In the case of Amoeba, we have limited ourselves to a straightforward implementation - the memory is transferred in its entirety after the process has been suspended, and before execution is restarted on the destination host. This is partly for pragmatic reasons - it is simplest to implement, and other methods depend on virtual memory, which has not been implemented for Amoeba. However, it should be noted firstly that the overhead of the more complex methods will only be worthwhile if a substantial proportion of the process's memory remains unreferenced. Secondly, lazy copying either imposes a residual dependency (as in Accent) which we wish to avoid, or requires that all dirty pages of the process be flushed to disk (as in Sprite), which imposes its own performance penalty. Thirdly, in Amoeba it is normal for a new process to be created on a different host (often an idle host) from the one used by the process requesting the creation. We feel that it is acceptable for the time taken by process migration to be comparable to that taken by remote process creation.

Given this decision, it is important that the overhead of memory transfer be kept low - the speed of transfer should be as close as possible to that which the networking hardware allows. Achievement of this aim is helped by the performance of the RPC mechanism as reported in [25]. However, it is necessary to take care that additional overhead is not imposed by the process migration mechanism. In particular, copying of large blocks of memory to and from RPC buffers should be avoided - the optimum solution is for RPC to transfer directly from the memory segment on the source host to the memory segment on the destination host. This is the design we decided on and reported in our previous paper [1]. We subsequently discovered that the *pro_exec* primitive in Amoeba 5 already does exactly this, and so no additional work was required.

5.2 Communication with a Migrating Process

The goal that migration should be transparent applies not only to the migrating process, but also to processes communicating with it. These processes should remain unaware of the migration, and be able to continue communication without any logical break. It *is* valid however for migration to cause communication delay - Amoeba is not a real-time operating system, and process migration is one of a number of potential causes of a communication time which is greater than normal.

The Amoeba communication mechanisms are RPC and group communications, both layered on a lower-level FLIP protocol, as described in section 3.1. There are no other input / output mechanisms in Amoeba; for example file operations are performed using RPC to a file server. At this stage we have restricted our migration implementation to dealing explicitly with RPC communications - group communications have been excluded. As far as we know there would be no significant new problems in dealing equally with the migration of processes engaged in group communication, but we have not spent the time required to investigate it.

It is necessary to consider the communication consequences of process migration both after migration of a process has completed (successfully or otherwise), and while a process is being migrated.

Communication after successful completion of migration: To avoid residual dependencies, communication after migration needs to be directly to and from the destination host, without relying on the source host to relay messages. This imposes two requirements: (a) the communication services for processes communicating with the migrated process must correctly route messages to the new host; (b) communication state must be migrated with the process.

In fact, the existing FLIP protocol was designed with process migration in mind, and already satisfies requirement (a). FLIP network addresses are *location-independent*, and FLIP uses a broadcast mechanism to find the host for a given address Normally this only happens the first time a process sends to a new address; the binding is then cached to avoid further broadcasts. However, when FLIP discovers that a cached binding is no longer valid, it repeats this procedure.

In our implementation, communication state is migrated as part of the extended checkpoint message produced by our migration variant of stun. An important item of communication state to be migrated is the FLIP address, of which there is one per process. The other items migrated define the state of communication being carried out by each thread of the process immediately before migration.

Communication during migration: Migration of a process takes a finite amount of time to complete. During this time, other processes may attempt to communicate with it on the source host, by sending a request message or returning a reply message. The process can deal with these messages only after completion of the migration. There are at least two ways of dealing with them:

- Queue the messages on the source and later transfer the queue to the destination, where they will be delivered when the process is restarted;
- Reject them and depend on the sender of the message to retransmit later.

The former method has the advantage of transparency, but can lead to substantial memory and communications overhead when there are large messages. It is also considerably more difficult to implement, given that Amoeba does not otherwise buffer messages.

For these reasons, the method chosen was the latter, using a *busy* status response to indicate that the process is temporarily unavailable to receive messages. This is not an error condition, and the sender is expected to handle this case by trying again later. Because the retry cannot lead to multiple delivery, it can safely be incorporated into the FLIP communication layer and is therefore completely transparent to application programs. This adds one message (FLIP_BUSY) to the FLIP protocol of [30]. Note that because it is implemented at the FLIP level, this technique applies equally to RPC request and reply messages.

Communication after failure of migration: Since process migration may fail for a variety of reasons, it must be possible for communication with the process to be reinstated normally when it resumes execution on its source machine. There is no need to handle this case specially - the mechanisms in the previous sections work equally well when the process resumes execution without having migrated.

6 User-Level versus Kernel-Level Implementation

The implementation as described so far involves changes to the kernel and the process server which runs in kernel mode. What remains is to control and coordinate the series of actions needed to migrate a process. This is the function of the *migration server*, which can operate as a user-level process. The migration server receives a migration request from a process executing some migration policy. It performs the requested migration by means of a sequence of RPC's with the process servers on the source and destination hosts. On completion (successful or otherwise), it replies to the migration request.

When performance tests are carried out, however, a problem becomes apparent with this approach. Process migration becomes slow when the source host has one or more compute-intensive processes in addition to the process to be migrated. These are of course just the conditions under which process migration is most likely.

This phenomenon is a somewhat subtle effect of the algorithm used by the Amoeba process scheduler. The scheduler gives priority to kernel-level processes, and uses a round-robin algorithm with fixed 100 ms timeslices for user processes. In effect, this favours compute-intensive processes (each of which always gets a full timeslice) over RPC-intensive processes (which generally do only a small amount of computing before being blocked). For the migration mechanism, the negative effect on performance comes about because much of the work performed in stunning a process for migration is done by code running as part of the user process itself; specifically, the generation and sending of the checkpoint message is done this way, and so the process to be migrated is, for a short period, RPC-intensive.

We investigated two approaches to this problem: change the scheduler, or move the migration server to the kernel. Changing the process scheduler allows the migration server to remain a user-level process. It also has the attraction that it is capable of improving all cases in which RPC-intensive processes compete with compute-intensive processes on one host. The change we made was to give higher priority (for no more than one timeslice) to a process thread when it becomes unblocked after transmission or reception of a FLIP packet. Experiments showed that this could greatly improve the performance of RPC-intensive processes. However, this solution introduces a change to the semantics of thread scheduling (threads within a process may use preemptive priority-based scheduling), and so we have not at this stage pursued this solution further.

Instead we have chosen to solve this problem by moving the remainder of the migration mechanism to the process server, which runs as a kernel-level thread and therefore has priority over user processes. This also has the advantage that the implementation is a little simpler (being confined to one server), and it improves efficiency by reducing the number of RPC's required. Our performance results (section 7) confirm that this solution is always faster than that based on a user-level migration server, and is much faster in the presence of compute-intensive processes.

7 Performance Results

7.1 Amoeba Results

All performance tests were carried out with Intel architecture PC's using the ISA bus and 3Com Etherlink II network adapters on an isolated Ethernet network operating at 10 Mbps. One 386 computer (33 Mhz) was used to run the file, directory and other ancillary Amoeba servers; three dedicated diskless 386 computers (40 Mhz) took part in the process migration experiments - one as the source host, one as the destination host, and the third for the migration server (where used). Elapsed times were measured using system call *sys_milli*, whose granularity (on PC's) is 50 ms.

Three artificial processes were created to allow us to observe the impact on migration time of various workloads on the source host. One is compute-intensive (an infinite loop). The other two are a client-server pair which perform null RPC's at maximum speed; when used these run on the same machine (in order not to affect network traffic) and execute approximately 100 RPC's per second.

The experiments measured how the speed of migration depends on process size, source host workload, and whether the migration server is in user-space or kernel-space. Experiments were done (i) with the source host idle, (ii) with one compute-intensive process on the source host; (iii) with a pair of RPC-intensive processes on the source host. These were done once with the user-space migration server (running on a third computer), and again with the migration mechanism incorporated into the process server, giving a total of six sets of results. In all cases the destination host was idle. All timing runs were performed ten times and averaged; the results proved highly repeatable, with occasional variations, attributed to periodic network activity by the Amoeba object manager, excluded from the averages.

The results are summarised in Figure 1. They show that in all cases the kernel solution is faster than the user-level solution. The difference is relatively small (approximately 300 ms) in most cases, but becomes large (around 1500 ms) when the source host has a compute-intensive process. The kernel-level solution is almost unaffected by the variation in source host workload. The reasons for this were discussed in section 6. Intriguingly, in the presence of RPC-intensive processes on the source host, migration is slightly faster than it is with an idle host. This is counter-intuitive; we have not investigated it, but it appears to be some subtle interaction between the RPC mechanism and the process scheduler.

For comparison, running the standard Amoeba test suite on our configuration gives a RPC throughput of 250 Kbytes per second (20% of raw Ethernet speed). Our time of approximately 4 seconds to migrate a 1 Mbyte process is therefore totally determined by RPC speed. We believe that the RPC speed in turn is limited by the (8-bit) network adapters we use - the Amoeba developers recently reported to us a RPC throughput of 1 Mbyte per second (80% of raw

Ethernet speed) using Sparc processors with fast network adapters.

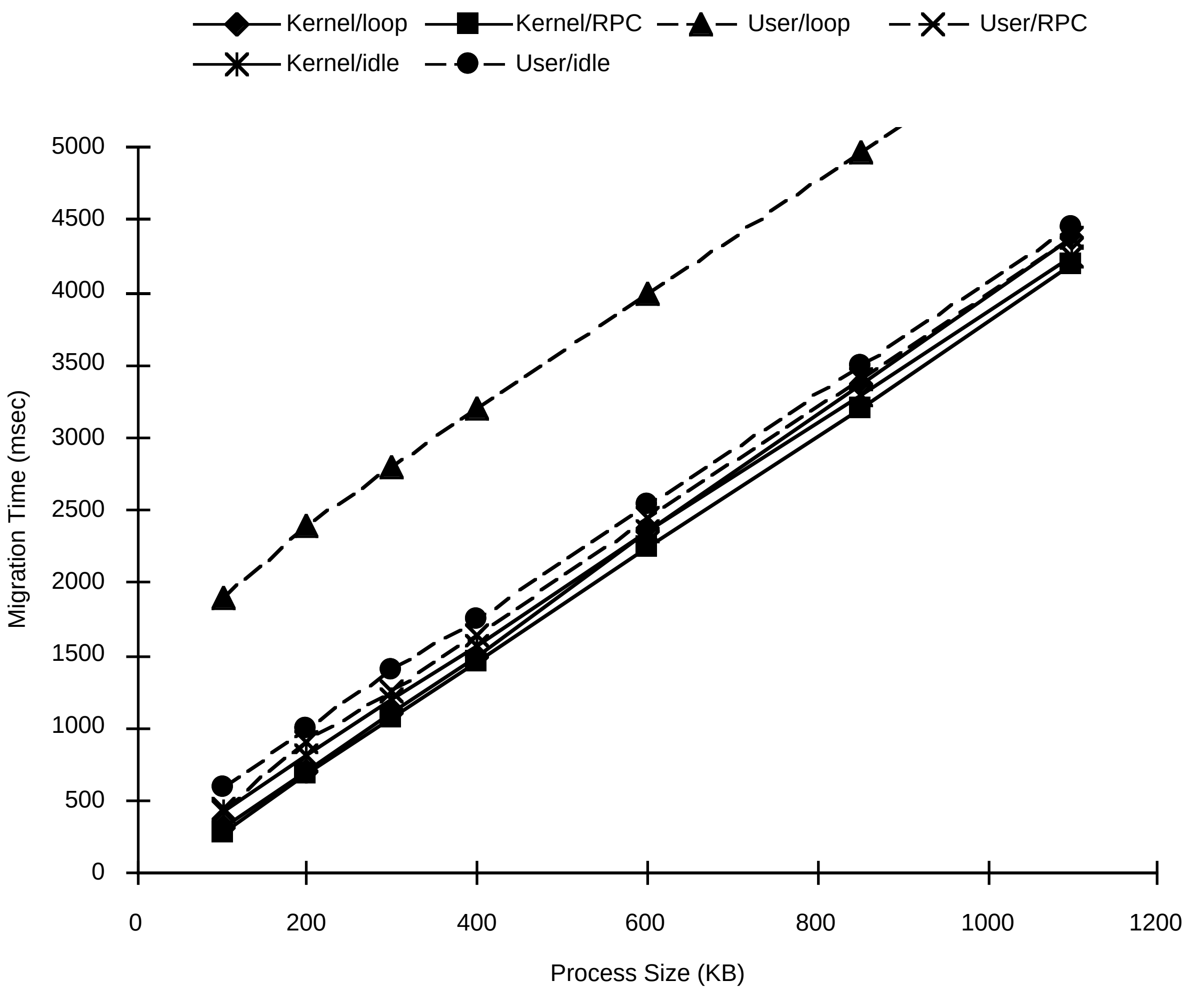

Figure 1: Process Migration times versus process size, source workload, and migration mechanism

7.2 Comparison with Other Implementations

Our results, for the kernel-space implementation on an idle source host, are well approximated by:
Amoeba: $T = 47 + 3.8m$
where T is the migration time in milliseconds, and m the process size in Kbytes.

The published performance of some other implementations is as follows:
Accent: $T = 1180 + 115v$
V: $T = 80 + 6v$
Charlotte: $T = 45 + 78p + 6.1v + 9.9r + 1.7q$
Sprite: $T = 76 + 9.4f + 0.48fs + 0.66v$
Mach: $T = 150 + 48n + 22.8rr + 5.5sr + 5.5sor + 58t$
where v is the virtual memory size of the process in Kbytes, f is the number of open files, fs is file size in Kbytes. (The other parameters are each specific to one implementation and their effect is minor.)

For 100 Kbyte processes, these typically amount to:
Amoeba: 430 ms; V: 650 ms; Charlotte: 750 ms; Sprite: 330 ms; Mach: 500 ms.

Not too much should be read into this comparison, as the tests were carried out at different times and based on different hardware. As already noted, the limiting factor in our implementation is the speed of the Ethernet adapters used, not the design of the mechanism itself; we would expect very significant improvements with faster networking hardware. No doubt improvements are also possible in the other implementations with improved hardware.

8 Summary and Conclusions

8.1 Review of Design Goals

In Section 4 we presented our design goals. Here we review to what extent these goals have been met.
Separation of policy and mechanism: This has been achieved by implementing the mechanism in a server and presenting a RPC interface to policy processes.
User-level implementation: Though it is possible to place part of the mechanism in a user-level process,

we concluded finally that it was better to place the mechanism completely in a kernel-level process for performance reasons; the conclusion might have been different if Amoeba had a process scheduler better suited to RPC-intensive processes.

Location transparency: As already discussed, we fall short of this goal in two respects. Firstly, we do not migrate group communication state, though we believe it will be straightforward to add this to our implementation. More seriously, migration is not completely transparent to a process migrated while blocked in a *trans* system call. More experience in the migration of a variety of processes is needed to assess how much this matters in practice.

Residual dependencies: Our process migration mechanism makes no use of residual dependencies.

Performance: Our implementation is limited only by networking speed; it remains to be seen to what extent this remains true for faster networks.

8.2 Lessons

Some lessons are to be learned from our experiences:

1 Our implementation shows that it is possible to achieve good performance from process migration using a careful but essentially straightforward design.

2 The principal difficulties with process migration are the encapsulation and migration of kernel state (including input/output state), and the redirection of inter-process communication. These are best dealt with by being designed into the system from the beginning, as in MOSIX [15] and RHODOS [34]. Failing that, a microkernel system offers the advantage of reduced kernel state and, often, network transparency. Even so, difficulties remain - none of the three microkernel implementations of which we are aware are completely satisfactory. The Mach implementation [17], like ours, aborts threads in kernel state and in addition leaves residual dependencies on the source host when migrating a Unix process. The implementation for Chorus [16] deals with system calls by waiting for them to complete. In the case of Amoeba, we believe a complete implementation is feasible, but requires more resources than we had available for this work.

3 More generally, our experience shows once again the effort required as outsiders to make a significant enhancement to a complex piece of software such as a distributed operating system. Though we have good access to the Amoeba team at Vrije Universiteit and have the complete documentation and source code, it took time to obtain the grasp of Amoeba internals required to design and implement our work. The implementation as described in this paper is the result of experience gained with two earlier prototypes.

4 Finally, the importance of experimental work in the study of distributed systems is reinforced. Many complex and subtle interactions between the components of a system are only found by building a proposed design and testing it - the unanticipated interaction between the process scheduler and the user-level migration server, though readily explainable after the event, is a small example of this.

8.3 Acknowledgments

We are grateful for the assistance of Andrew Tanenbaum and the members of the Amoeba project, past and present, in making Amoeba available and in providing support and information. Greg Sharp in particular deserves special thanks for his continuing patience and prompt responses to questions. Thanks are also due to the School of Computer and Information Science at the University of South Australia and the Stefan Batory Foundation in Warsaw for helping to fund the visit of two of us to the University of South Australia, and to the Institute of Informatics at Warsaw University for facilitating the visit and for the use of its resources.

9 References

[1] C.F. Steketee, W.P. Zhu and P.A. Moseley, “Implementation of Process Migration in Amoeba”, in *Proc. 14th International Conference on Distributed Computing Systems*. Poznan, Poland, IEEE Computer Society Press, pp. 194-201, 1994.

[2] W.P. Zhu, C. Steketee and B. Muilwijk, “Load balancing and workstation autonomy on Amoeba”. *Australian Computer Science Communications*. vol. 17, pp. 588-597, 1995.

[3] W.P. Zhu and C.F. Steketee, “An Experimental Study of Load Balancing on Amoeba”, in *Proc. Aizu International Symposium on Parallel Algorithms / Architecture Synthesis*. Aizu-Wakamatsu, Japan, IEEE Computer Society Press, pp. 220-226, 1995.

[4] D.L. Eager, E.D. Lazowska and J. Zahorjan, “The Limited Performance Benefits of Migrating Active Processes for Load Sharing”, in *Proc. ACM SIGMETRICS 1988*. pp. 63-72, 1988.

[5] W.E. Leland and T.J. Ott, “Load-balancing Heuristics and Process Behavior”, in *Proc. PERFORMANCE'86 and ACM SIGMETRICS 1986*. 1986.

[6] P. Krueger and M. Livny, “A Comparison of Preemptive and Non-Preemptive Load Distributing”, in *Proc. 8th International Conference on Distributed Computer Systems*. 1988.

[7] F. Douglis and J. Ousterhout, “Transparent Process Migration: Design Alternatives and the Sprite Implementation”. *Software - Practice and Experience*. vol. 21, no. 8, pp. 757-785, 1991.

[8] M.L. Powell and B.T. Miller, “Process Migration in DEMOS/MP”, in *Proc. 9th ACM Symposium on Operating System Principles*. 1983.

[9] Y. Artsy and R. Finkel, “Designing a Process Migration Facility: the Charlotte Experience”. *Computer*. vol. 22, no. 9, pp. 47-56, 1989.

[10] M.M. Theimer, K.A. Lantz and D.R. Cheriton, “Preemptable Remote Execution Facilities for the V-System”, in *Proc. 10th Symposium on Operating System Principles*. pp. 2-12, 1985.

[11] E. Zayas, “Attacking the Process Migration Bottleneck”, in *Proc. 11th ACM Symposium on Operating Systems Principles*. Austin, TX, ACM, pp. 13-22, 1987.

[12] G. Thiel, “LOCUS Operating System, a Transparent System”. *Computer Communications*. 1991.

[13] P.K. Sinha, “Process Migration in the GALAXY Distributed Operating System”, in *Proc. 5th International Parallel Processing Symposium*. pp. 611-618, 1991.

[14] M. Litzkow and M. Solomon, “Supporting Checkpointing and Process Migration Outside the UNIX Kernel”, in *Proc. USENIX Winter Conference*. San Francisco, pp. 283-290, 1992.

[15] A. Barak, S. Guday and R.G. Wheeler, “The MOSIX Distributed Operating System: Load Balancing for Unix”, *Lecture Notes in Computer Science 672*. Springer-Verlag, 1993.

[16] M. O'Connor, B. Tangney, V. Cahill and N. Harris, “Microkernel Support for Migration”. *Distributed Systems Engineering Journal*. 1993.

[17] D. Milojicic, W. Zint, A. Dangel and P. Giese, “Task Migration on the Top of the Mach Microkernel”, in *Proc. MACH III Symposium*. Sante Fe, NM, USENIX, pp. 273-289, 1993.

[18] H. Schrimpf, “Migration of Processes, Files, and Virtual Devices in the MDX Operating System”. *Operating Systems Review*. vol. 29, pp. 70-81, April 1995.

[19] J.M. Smith, “A Survey of Process Migration Mechanisms”. *ACM Operating System Review*. vol. 22, no. 3, pp. 28-40, 1988.

[20] M.R. Eskicioglu, “Design Issues of Process Migration Facilities in Distributed Systems”. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*. vol. 4, no. 2, 1990.

[21] M. Nuttall, “A brief survey of systems providing process or object migration facilities”. *Operating Systems Review*. vol. 28, no. 4, pp. 64-80, Oct. 1994.

[22] M.R. Eskicioglu and L.-F. Cabrera, “Process Migration: An Annotated Bibliography”. *IEEE Computer Society Technical Committee on Operating Systems Newsletter*. vol. 4, no. 4, 1990.

[23] H.E. Bal, A.S. Tanenbaum and M.F. Kaashoek, “Experiences with Distributed Programming in Orca”, in *Proc. IEEE CS International Conference on Computer Languages*. New Orleans, Louisiana, 1990.

[24] H.E. Bal, A.S. Tanenbaum and M.F. Kaashoek, “Orca: A Language for Distributed Programming”. *SIGPLAN Notices*. vol. 25, no. 5, pp. 17-24, 1990.

[25] A.S. Tanenbaum, *et al.*, “Experiences with the Amoeba Distributed Operating System”. *Communications of the ACM*. vol. 33, no. 12, 1990.

[26] A.S. Tanenbaum and R. van Renesse, “Distributed Operating Systems”. *Computing Surveys*. vol. 17, no. 4, pp. 419-470, 1985.

[27] R. van Renesse, A.S. Tanenbaum and A. Wilschut, “The Design of a High-Performance File Server”, in *Proc. 9th International Conference on Distributed Computer Systems*. IEEE, pp. 22-27, 1989.

[28] R. van Renesse, “The Functional Processing Model”. PhD thesis, Vrije Universiteit, Amsterdam, 1989.

[29] A.D. Birrell and B.J. Nelson, “Implementing Remote Procedure Calls”. *ACM Transactions on Computer Systems*. vol. 2, no. 1, pp. 39-59, 1984.

[30] M.F. Kaashoek, R. van Renesse, H. van Staveren and A.S. Tanenbaum, “FLIP: An Internetwork Protocol for Supporting Distributed Systems”. *ACM Transactions on Computer Systems*. vol. 11, no. 1, pp. 73-106, 1993.

[31] M.M. Theimer and B. Hayes, “Heterogeneous Process Migration by Recompilation”, in *Proc. 11th International Conference on Distributed Computing Systems*. pp. 18-27, 1991.

[32] G. Sharp (ed.) *Amoeba Reference Manual*, Vrije Universiteit en Stichting Mathematisch Centrum, 1995.

[33] S.J. Mullender, “Process Management in a Distributed Operating System”, in *International Workshop on Experiences with Distributed Systems*, Lecture Notes in Computer Science 309. Springer-Verlag, 1989.

[34] W. Zhu and A. Goscinski, “The Development of the Load Balancing Server and Process Migration Manager for RHODOS”. Technical Report CS90/47, Department of Computer Science, University College, University of New South Wales, 1990.