Process Migration and Load Balancing in Amoeba

Chris Steketee

Advanced Computing Research Centre,
School of Computer and Information Science,
University of South Australia,
The Levels SA 5095
Email: *Chris.Steketee@cis.unisa.edu.au*

Abstract. This paper reports our experience in adding process migration to the distributed operating system Amoeba, and the results of a series of experiments to evaluate its usefulness for load balancing. After describing our design goals, we present our implementation for Amoeba, and performance figures which indicate that the speed of process migration is limited only by the throughput of the network adapters used in our configuration. We also present load balancing results showing that process migration can make a substantial improvement to the performance of a distributed system.

1 Introduction

This paper describes our development of a process migration mechanism for the distributed operating system Amoeba, and the results of experiments to evaluate its usefulness for load balancing. In addition, we make some comments on the lessons we have learnt.

In previous papers, we presented the design of a process migration mechanism for Amoeba, giving the results of a prototype implementation [Steketee et al., 1994; Steketee et al., 1996], and reported the results of preliminary load balancing studies using the prototype [Zhu et al., 1995; Zhu and Steketee, 1995]. The conclusions of these studies 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. Since then, we have completed a full implementation of process migration based on the same design, and have carried out 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

Proceedings of the Twenty Second Australasian Computer Science Conference, Auckland, New Zealand, January 18--21 1999.

communication. This paper presents the new implementation and its performance, and follows this with the results of the load balancing experiments.

First, a few definitions. A **distributed system** is a set of autonomous computers called **hosts**, communicating via a network and cooperating to achieve some common goal. A **distributed operating system** is an operating system which controls and allocates the resources of a distributed system. A **process** is a program in a state of execution; it may be multi-threaded, however, it resides entirely on one host. **Process migration** is the movement of a running process from one host to another. **Load balancing** is the assignment of processes to hosts with the aim of achieving an even distribution of load. In **static load balancing**, processes are assigned to hosts when first created and remain on that host for their lifetime, whereas **dynamic load balancing** allows processes to be migrated subsequently in order to correct a load imbalance.

We consider a distributed system to consist of a set of homogeneous hosts, which, in the absence of other factors, are all equally suitable candidates for initial placement of a process, and equally suitable destinations for process migration. This contrasts with some other studies, where the emphasis is on personal workstations and the temporary migration of processes to workstations which are idle.

2 Overview of Amoeba

Amoeba is a research distributed operating system developed at Vrije Universiteit, Amsterdam over the period 1981 to 1996. The last version to be developed, Amoeba 5, runs on Intel 80x86, Motorola 680x0, and SPARC platforms. An excellent exposition of Amoeba can be found in [Tanenbaum et al., 1990]. A few aspects of the Amoeba design, sufficient for the purposes of this paper, are briefly summarised below.

Microkernel design: The Amoeba kernel is small and maintains relatively little process state. In particular, the state of open files is completely maintained by user processes, not the kernel, and can be migrated with a process.

Interprocess communication: Amoeba's basic model for inter-process communication is the *Remote Procedure Call* or *RPC.* 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*. In addition to RPC, there are multicast and atomic group communication facilities. All communication is layered on top of the FLIP network protocol [Kaashoek et al., 1993].

Location transparency: Amoeba provides location transparency - neither end-users nor application programs need to know the network location of processes or other objects.

3 Process Migration

Process migration has been the subject of a considerable amount of research, and there have been a number of implementations reported in the literature, both for distributed operating systems, eg [Theimer et al., 1985; Douglis and Ousterhout, 1991; Thiel, 1991; O'Connor et al., 1993; Milojicic, 1994] and for Unix, eg [Litzkow and Solomon, 1992; Barak et al., 1996]. Motivations for process migration include load balancing, and locality - the ability of a process to move to the same host as some resource or user. Our main interest in process migration is to assess experimentally its applicability to load balancing.

3.1 Implementation of Process Migration for Amoeba

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.

First we present our design goals. More detail on the design is to be found in [Steketee et al., 1994] and [Steketee et al., 1996].

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 – an RPC with arguments P - the process to be migrated, S - the source host, and D - the destination host.

Location transparency: Users, and user processes, should not be concerned with where processes run; 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 required 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, in order to maximise its usefulness for load balancing.

3.2 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.

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 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 migrating the process. The problem arises with blocking system calls. It is not satisfactory to wait until these complete, since the delay can be indefinitely long.

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 may be possible in principle, it is a task we were unwilling to attempt with the time and resources available to us. We therefore chose to abort blocked system calls when a process is to be migrated. 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 as a side-effect of having been migrated.

The main potential problem is where a RPC *transaction* 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. Similar effects occur when a server is migrated while blocked in the system call that sends a reply.

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, for example, in Sprite [Douglis and Ousterhout, 1991], in which pages of the process address space are moved to the destination host only when referenced.

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. There are several reasons for this. Firstly, the overhead of more complex methods is only worthwhile if a substantial proportion of the process's memory remains unreferenced. Secondly, lazy copying either imposes a residual dependency, or requires that all dirty pages of the process be flushed to disk. Thirdly, in Amoeba it is the norm 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, and we feel that it is acceptable for the time taken by process migration to be comparable to that taken by remote process creation. Lastly, it is far simpler to implement.

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 [Tanenbaum et al., 1990]. It is necessary also 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 must be avoided - network transfers should be directly from the memory segment on the source host to its final location on the destination host.

3.3 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 be able to continue communication without any logical break.

The Amoeba communication mechanisms are RPC and group communications, both layered on a lower-level FLIP protocol. There are no other input / output mechanisms in Amoeba; for example file operations are performed using RPC to a file server. We have restricted our migration implementation to dealing explicitly with RPC communications - inclusion of group communication, while necessary for a production implementation, would not elicit new research issues.

Communication after successful completion of migration: To avoid residual dependencies, communication after migration needs to be directly with the new host, without relying on the old 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. With respect to (a), this is satisfied by the FLIP protocol - FLIP network addresses are *location-independent* and FLIP caters for their migration. (b) is part of our migration implementation.

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. So the method chosen was the latter, using a *busy* status response to indicate that the process is temporarily unavailable to receive messages. The sender is expected to handle this case by trying again later. This has been incorporated into the FLIP communication layer and is therefore completely transparent to application programs. It adds one message (FLIP_BUSY) to the FLIP protocol of [Kaashoek et al., 1993].

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.

3.4 User-Level versus Kernel-Level Implementation

The implementation as described so far involves changes to the Amoeba kernel. 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 RPCs with the kernels on the source and destination hosts. On completion (successful or otherwise), it replies to the migration request.

In practice, the performance of a user-level migration server suffers 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 problem arises from a shortcoming of the Amoeba process scheduler and is discussed in detail in [Steketee et al., 1996]. Although we had some success in overcoming this with an improved process scheduler, it required changing some of the semantics of thread scheduling, and we did not persist with this approach.

Instead we chose 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 and reduces the number of RPCs required. Our performance results (section 4) 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.

4 Performance Results for Process Migration

All performance tests were carried out with Intel architecture PCs 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). Experiments were done (i) with the source host idle and (ii) with a compute-intensive process 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 four sets of results. In all cases the destination host was idle. All timing runs were performed ten times and the results averaged.

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. RPC throughput with our configuration is 250 Kbytes per second (20% of raw Ethernet speed), so our time of approximately 4 seconds to migrate a 1 Mbyte process is totally determined by RPC speed. The RPC speed in turn is largely limited by the (8-bit) network adapters used on our ancient PCs - the Amoeba developers reported a RPC throughput of 1 Mbyte per second (80% of raw Ethernet speed) using Sparc processors with fast network adapters.

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

The comparison with published performance figures for other implementations, using 100 Kbyte processes, is: Amoeba: 430 ms; V: 650 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.

5 Load Balancing

Load balancing is the distribution of processes amongst the hosts of a distributed system in order to equalise the load amongst them.

The most important technique available for load balancing is *process placement* - the initial allocation of a newly-created process to a suitable host. Perfect process placement would choose a host which maximises the desired performance criteria over the lifetime of that process. In practice, the future behaviour of a process cannot in general be predicted, and so practical process placement is in most cases limited to maximising the performance at the instant of process creation - typically by choosing the processor most lightly loaded at the time. This can cause subsequent imbalance, for example when all the processes on some computers terminate while leaving others heavily loaded. Even then, such an imbalance may not matter: if the workload consists entirely or predominantly of a steady flow of short-lived processes, then process placement will soon correct the imbalance. On the other hand, if the workload consists largely of long-running compute-intensive processes, long-term imbalance is likely.

This is the reason for the interest in process migration as an additional load balancing technique - the movement of running processes from heavily loaded hosts to lightly loaded ones can correct long-term imbalance. Process migration does however have significant overheads in comparison with initial process placement. The challenge therefore is to devise algorithms which undertake process migration only when it is likely to improve net performance.

Load balancing has been studied extensively by simulation. The conclusions, particularly for dynamic load balancing, vary. Eager, for example, concludes that process migration does not provide a significant improvement [Eager et al., 1988], whereas others [Krueger and Livny, 1988; Hac, 1989] have come to the opposite conclusion. By contrast, there have been few experimental studies [Milojicic, 1994; Barak et al., 1996].

Our aim has been to carry out simple experimental studies on Amoeba using synthetic workloads comparable to those in simulation studies. Our first experiments [Zhu et al., 1995; Zhu and Steketee, 1995] indicated that the benefits of migration were marginal. However, these results were affected by the poor performance of the prototype migration mechanism used. The next section presents the results of repeating these experiments with the full migration mechanism.

5.1 Implementation of Load Balancing Experiments

Our load balancing facility consists of processes of several kinds. Firstly, a *load balancer* implements the placement and/or migration policy being studied. This uses system calls for process creation and to invoke the process migration mechanism.

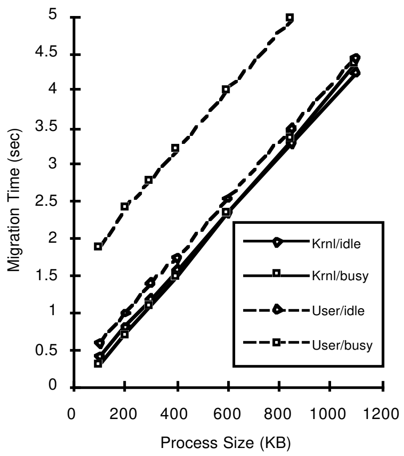

Fig. 1. Process Migration time

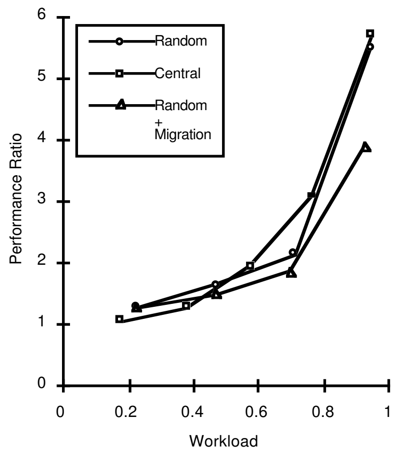

Fig. 2. Load Balancing for 2 Hosts

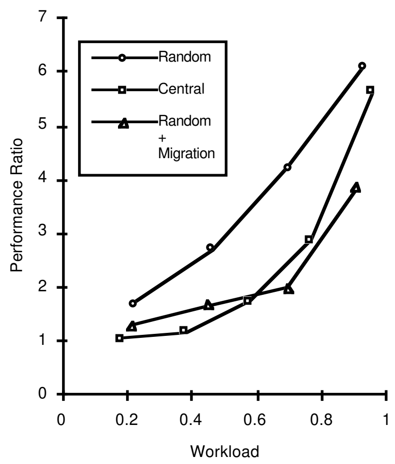

Fig. 3. Load Balancing for 4 Hosts

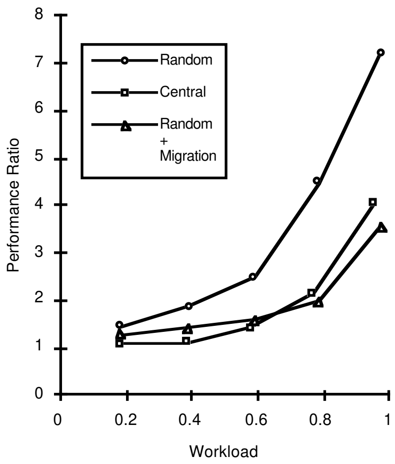

Fig. 4. Load Balancing for 6 Hosts

Secondly, a *workload generator* produces a series of *worker processes*, whose interarrival time follows a Poisson distribution and whose service time (application CPU time) is exponentially distributed. The parameters of the time distributions are variable. Once started, each worker process executes a loop to consume its allotted service time. Worker processes carry out no communication. For these experiments, the mean service time was fixed at 5 seconds, and the worker process memory at 100 KB. The interarrival time was varied in order to produce the required workload.

We use a set of identical diskless hosts in our experiments, plus a file server for reading executables and storing results (see 4.1). We dedicate additional hosts to the workload generator and the load balancer, and to a statistics server which collects results.

5.2 Load Balancing Algorithms

Our experiments compared three load balancing algorithms:

Random placement: A new process is created on a randomly chosen host.

Central placement: A new process is created on the host which has the lowest load, when last measured.

Random placement plus central migration: A new process is created on a randomly chosen host. When a sufficiently large load imbalance is detected, one or more processes are migrated. For these experiments, we regard a host as overloaded if it has more than two worker processes and underloaded if it has zero worker processes, and migrate processes from overloaded to underloaded hosts.

Note that there is an obvious fourth algorithm to add to these - central placement plus central migration. From our previous results with the prototype migration mechanism, as well as by extrapolation from the results of the other three algorithms, we would expect this algorithm to show a significant improvement over central placement for high workloads.

5.3 Performance Results

As a *performance index*, we use the ratio of mean response time to mean service time, where response time is the time elapsed between the creation of a process and its completion. A performance index of 1.0 indicates perfect performance, which is only possible when each process has a dedicated host and overheads are small. It will be noted from the figures below that the performance index is gratifyingly close to 1.0 for low to moderate workloads.

To measure the load on the collection of hosts running worker processes, we use the *workload ratio*, defined as the ratio of mean service time to mean interarrival time, multiplied by the number of hosts. A value of 0 means idle; values approaching 1 indicate a fully loaded system.

Figures 2 to 4 compare the three algorithms for 2, 4 and 6 hosts. It is clear that random placement performs badly with increasing workload, as is to be expected, and that both central placement and central migration improve significantly on this. It is encouraging that central migration always improves on random placement, even at low workloads, and that it outperforms central placement at high workloads, successfully overcoming the poor decisions made by random placement.

6 Summary and Conclusions

6.1 Review of Design Goals for Process Migration

In Section 3 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.

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

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

Performance: Limited only by networking speed in our current configuration.

6.2 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 [Barak et al., 1996] and RHODOS [Zhu and Goscinski, 1990]. Failing that, a microkernel system offers the advantage of reduced kernel state and, in the case of Amoeba, network transparency. Even so, difficulties remain - none of the three microkernel implementations of which we are aware are completely satisfactory. The Mach implementation [Milojicic, 1994], 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 [O'Connor et al., 1993] deals with system calls by waiting for them to complete. In the case of Amoeba, a complete implementation should be feasible, but requires more resources than we had available for this work.

Acknowledgments

We are grateful for the assistance of Andrew Tanenbaum and the members of the Amoeba project in making Amoeba available and in providing support and information. Thanks are due also to Weiping Zhu for contributing his experience on load balancing and for the earlier experimental results. A number of University of South Australia students worked on this project, including some visiting students from Holland and Poland, who did much of the hard programming and experimental work and gave up many nights' sleep in order to find a set of idle PCs for experiments. Our thanks to all of them.

References

Barak, A. et al. (1996). Performance of PVM with the MOSIX Preemptive Process Migration. In *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering. Herzliya*. pp. 38-45.

Douglis, F. and Ousterhout, J. (1991). Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience* **21**(8). pp. 757-785.

Eager, D.L. et al. (1988). The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In *Proc. ACM SIGMETRICS 1988*. pp. 63-72.

Hac, A. (1989). A Distributed Algorithm for Performance Improvement through File Replication, File Migration and Process Migration. *IEEE Trans. on Software Engineering* **15**(11).

Kaashoek, M.F. et al. (1993). FLIP: An Internetwork Protocol for Supporting Distributed Systems. *ACM Transactions on Computer Systems* **11**(1). pp. 73-106.

Krueger, P. and Livny, M. (1988). A Comparison of Preemptive and Non-Preemptive Load Distributing. In *Proc. 8th International Conference on Distributed Computer Systems*.

Litzkow, M. and Solomon, M. (1992). Supporting Checkpointing and Process Migration Outside the UNIX Kernel. In *Proc. USENIX Winter Conference. San Francisco*. pp. 283-290.

Milojicic, D.S. (1994). *Load Distribution: Implementation for the Mach Microkernel.* Wiesbaden, Verlag Vieweg.

O'Connor, M. et al. (1993). Microkernel Support for Migration. *Distributed Systems Engineering Journal*.

Steketee, C.F. et al. (1996). Experiences with the Implementation of a Process Migration Mechanism for Amoeba. *Australian Computer Science Communications* **18**(1). pp. 140-148.

Steketee, C.F. et al. (1994). Implementation of Process Migration in Amoeba. In *Proc. 14th International Conference on Distributed Computing Systems. Poznan, Poland*. pp. 194-201. IEEE Computer Society Press.

Tanenbaum, A.S. et al. (1990). Experiences with the Amoeba Distributed Operating System. *Communications of the ACM* **33**(12).

Theimer, M.M. et al. (1985). Preemptable Remote Execution Facilities for the V-System. In *Proc. 10th Symposium on Operating System Principles*. pp. 2-12.

Thiel, G. (1991). LOCUS Operating System, a Transparent System. *Computer Communications*.

Zhu, W. and Goscinski, A. (1990). The Development of the Load Balancing Server and Process Migration Manager for RHODOS. Department of Computer Science, University College, University of New South Wales.

Zhu, W.P. et al. (1995). Load balancing and workstation autonomy on Amoeba. *Australian Computer Science Communications* **17**(1). pp. 588-597.

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