Implementation of Process Migration in Amoeba

Chris Steketee, Wei Ping Zhu, and Philip Moseley

School of Computer and Information Science, University of South Australia, The Levels SA 5095, Australia. Chris.Steketee@Unisa.edu.au

Abstract

The design of a process migration mechanism for the Amoeba distributed operating system is described. The primary motivation for this implementation is to carry out experimental and realistic studies of load balancing algorithms for a distributed operating system. Our aim has been the implementation of a mechanism which is general, efficient and fully transparent, and which is reliable in the presence of network and processor failures.

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 an implementation of process migration for the distributed operating system Amoeba.

Our primary interest in process migration is to give us a platform for carrying out experimental studies of load balancing algorithms. A secondary interest is to investigate its applicability to fault-tolerance, but this will not be further expanded upon in this paper. Process migration as a means of achieving load balancing has been the object of a number of studies. The conclusions of these studies have been equivocal [1-3], and there is scope for further work to clarify its usefulness and applicability.

Recently, one of us carried out simulation studies of a range of load-balancing algorithms with a view to comparing their relative performance [4]. The results show that source-initiated algorithms for load balancing outperform server-initiated algorithms in almost all cases. This agrees with the findings of [5] but is contrary to the conclusions of some researchers (for example [6]) that server-initiated algorithms perform better than source-initiated algorithms under many circumstances.

It is clear that there is potential for further study both of the usefulness of process migration for load balancing, and the relative merits of the various algorithms. In particular, experimental studies in this field are still uncommon, and would form a valuable complement to the various simulation results available. It is with this motivation that we decided to implement a process migration facility.

We chose to use the distributed operating system Amoeba [7] as the platform for our work for two principal reasons. Firstly, Amoeba appeared well-suited by virtue of several properties, including location-independent addressing and its micro-kernel design with an emphasis on the provision of services by user-level processes. Secondly, availability: Amoeba is well-documented in the literature, the implementation, including source code, is readily available to Universities, and useful work can be done on a basic equipment configuration which does not require major expenditure.

This paper reports on the design and implementation of the process migration mechanism. The use of this mechanism for experimental studies of load-balancing is the subject of ongoing work, the results of which will be reported separately.

The remainder of this paper is organised as follows. In Section 2 we give a brief overview of process migration, experiences, motivation, and implementation. Section 3 describes Amoeba, concentrating on those aspects most relevant to our work. Section 4 is concerned with design issues for process migration mechanisms and Section 5 gives details of our design. This is followed in Section 6 by a summary of the current status of our work, and discussion of possible areas for further work.

2 Overview of process migration

The term *process migration* means the movement of an executing process from one host processor (the *source*) in a distributed computing system to another host (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 have been 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 experimental implementations reported in the literature [8-14]. The potential benefits of process migration have been classified [15, 16] as:

Load balancing - improved performance for a distributed computing system overall, or a distributed application, by spreading the load more evenly over a set of hosts;

Reduction in communication overhead - by locating on one host a group of processes with intensive communication amongst them;

Resource access - not all resources are available across the network: a process may need to migrate in order to access a special device, or to satisfy a need for a large amount of physical memory;

Fault-tolerance - allowing long running processes to survive the planned shutdown or failure of a host.

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, deleting the process on the source, and resuming the process's execution on the destination. 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 machine. 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 route correctly messages sent to the process during and after migration. These issues are taken up in more detail in Sections 5 and 6.

A more detailed overview of process migration design issues may be found in [16], and an annotated bibliography in [17].

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. Although primarily intended for research purposes, it is well-developed for use as a software development platform. It has a Unix emulation facility, and includes support for TCP/IP, the X-window system, MMDF mail, and Tex. In the design and development of Amoeba there has been a strong emphasis on performance and on conceptual simplicity of design. 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 [18, 19].

An excellent exposition of Amoeba can be found in [7]. In this section, we summarise the concepts of Amoeba, concentrating on those aspects which are most relevant to this paper, including developments since [7].

3.1 Amoeba concepts

Transparency: Amoeba is a distributed operating system as defined in [20]. It provides location transparency - neither end-users, nor application programs, need to know the location of the objects (including processes) being addressed.

Objects, servers and capabilities: Amoeba is an *object-based* operating system. Objects are managed by *servers*, one for each type of object. An object is identified by a *capability*, consisting of a *port* and a *private part*. The port identifies the server managing this object type; the private part is used by the server to identify and protect the specific object. In order to perform an operation on an object, it is necessary to have a capability for that object. Amoeba has a number of standard servers, including the Bullet file server [21], the Soap directory server [22], a process server and a time-of-day server. Servers are also written by users when developing new applications.

Remote procedure call: Amoeba's mechanism for inter-process communication is the *Remote Procedure Call* or *RPC* [23]. This is implemented using synchronous message passing - the client process sends a *request* message to the server, which carries out the request and responds with a *reply* message.

RPC uses 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. The Amoeba kernel performs no buffering of request or reply messages - this avoids message copying and therefore speeds performance.

A request is addressed to a port, which is a location-independent permanent identifier for a server, normally embedded in the capability for the object being accessed. The Amoeba kernel on the client host broadcasts a *locate* message to find the server which has an outstanding getreq for that port. As an important optimisation, client kernels maintain a cache of server locations, thus eliminating the need for most locate broadcasts.

Fast local internet protocol (FLIP): Starting with Amoeba 5, the RPC mechanism is layered on top of a new network protocol, FLIP [24]. This provides processes with location-independent network addresses, which have been designed with process migration in mind, since they 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 locate processes, 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 - Amoeba does not use virtual memory, to avoid its performance impact, and on the principle that physical memory is relatively cheap. 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.

Micro-kernel design: The Amoeba kernel contains only those functions which need to be there, with other 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. Other operating system functions are carried out by servers.

Process server: The functions of process management in Amoeba are performed by a *process server* which runs in each host. The process server in turn makes use of system calls to the kernel in order to carry out some of its functions, such as memory allocation. This division of labour ensures that the normal functions required by user programs, such as process creation, can be invoked from a client running on a remote host, by using RPC transactions with the appropriate process server. The process server runs in kernel mode because of its need to carry out privileged functions.

Run server: Amoeba has the ability to perform static load balancing by means of process placement, in which the most lightly-loaded suitable host is selected when a new process is to be created. This is based on processor load statistics collected regularly from each host. The collection of statistics, and selection of the host, are carried out by the *run server*, which is described in [25].

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:

- Amoeba's communication mechanisms, particularly the FLIP layer in Amoeba 5, are location-transparent;
- The micro-kernel design means that the kernel keeps relatively little process state; in particular, files and devices are accessed by means of transactions to servers.

It should also be noted that the addition of process migration to Amoeba has been canvassed on at least two previous occasions to our knowledge [26, 27]. While these plans were not in fact carried through, they have been valuable precursors. The former paper describes a mechanism for process checkpoints, including the suspension of process threads in a clean state (see 5.1 below), which was implemented and has provided some of the infrastructure needed for process migration. The latter paper includes a design for process migration which gave us considerable insight, though our final design differs substantially from that one. In particular, ideas taken over from the latter paper include the generation of a "migration pending" status response to communication with a migrating process, and the use of the location-independence of FLIP addresses (see 5.2). The most significant differences between that design and ours are the handling of communication issues at the FLIP level thereby avoiding the asymmetry between client and server inherent in RPC, and the avoidance of memory-memory copying in transferring the memory state of a process (5.3 step 4). The emphasis on fail-safe migration (5.4) is also new in our work.

4 Design issues for a process migration mechanism

Our particular interest in process migration at this time is its application to load balancing. In order to carry out experimental studies of load balancing in a realistic environment, we wish to place as few restrictions as possible on our implementation. Ideally it should be possible to migrate an arbitrarily-chosen process to an arbitrarily-chosen host without the knowledge or involvement of either the migrant process or those processes with which it is communicating.

4.1 Homogeneous versus heterogeneous migration

Our only major restriction is to limit ourselves to *homogeneous* migration, in which processes are migrated between processors of the same architecture. While there has been some work on heterogeneous migration, for example [28], 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[1].

4.2 Separation of policy from mechanism

Our implementation, like some others, takes care to separate process migration *policy* from process migration *mechanism*. The mechanism is concerned with *how* migration is carried out, and a substantial part is by necessity implemented in kernel processes. Alterations to the mechanism require recompilation of the kernel, and errors are likely to result in failure of the host. By contrast, process migration policy is concerned with *when* and *where* to migrate *which* process.

In order to allow ready experimentation with process migration policies, whether for load balancing or other purposes, the policies are implemented in normal user-level processes[2]. Policy changes do not require recompilation of the kernel, and the effects of errors are less drastic. Moreover, this approach allows a range of policies to be studied, from completely centralised (one system-wide process implementing migration policy) to completely distributed (one policy process per host).

Our implementation effects this separation by means of a *process migration server* process which initiates and coordinates migration. A copy of this server runs on each host, and the migration is carried out using transactions with the process server on each host and also between the process migration servers on the source and destination. We chose this approach, rather than placing our implementation completely within the existing process server, for reasons of modularity.

[1] Note that, by contrast, heterogeneous process *placement* is quite feasible and has been implemented, for example, in the Amoeba. run server

[2] At present, the "policy" processes are essentially trivial ones for testing the mechanism - eg migrate a specific process to a specific host as specified by user input.

4.3 Security

The fact that process migration is requested by user-level processes means that it is potentially available to an arbitrary user. In some cases this would be an undesirable state of affairs as error or malice could lead to a grossly-imbalanced system. We therefore allow a system administrator to restrict access to the process migration mechanism or to leave it unrestricted. This is achieved by requiring the requesting process to present a process migration capability to the process migration server - this capability is stored in the directory server, and the ability to restrict access is available using the normal mechanisms of that server.

4.4 Transparency

An important goal in process migration is transparency. This means that neither the process being migrated, nor user processes with which it is communicating, should be aware of the migration. Our design has aimed for this degree of transparency in the implementation of Amoeba process migration, and has essentially achieved it completely. The exceptions are a few low-level system calls which are host-dependent, for example one which returns the value of the hardware clock on the local host[3]. The only other detectable effect of the migration of a process is timing, in particular the execution hiatus when a process is migrated.

4.5 Residual dependencies

A particular problem in migrating a process is the routing of messages addressed to the migrated process, since the sender of the message need not know about the migration. One way of handling this is for the *source* machine to redirect messages to the *destination* machine. This is an example of a *residual dependency*. In general residual dependencies are undesirable because of the chain of dependencies when a process is migrated several times and the continuing use of resources on the source machine. This has detrimental effects on both performance and reliability.

Our implementation makes no use of residual dependencies. In particular, message routing is achieved using the inherent location-independence of Amoeba communication protocols.

[3] The normal way of accessing time is to use a system-wide Time-of-Day server.

4.6 Memory transfer

Typically the greatest cost of migrating a process is the time spent copying the memory image of the process from source to destination machine, since this is limited by communication speeds. Various approaches have been adopted to this. In a straightforward implementation, the process is first frozen on the source, then the complete memory image is transferred to the destination, and finally process execution is resumed on the destination. This is the approach used by Charlotte [12] and LOCUS [11]. V [13] uses *pre-copying* of memory pages in parallel with continued execution of the process on the source machine. While this in fact increases the total amount of work, since pages modified during this period have to be copied twice, it does greatly reduce the *migration latency*, that is the time during which the process is frozen.

Accent [14] by contrast uses *lazy copying*, in which pages are moved to the destination from the source host only when referenced. The advantage is that often a substantial proportion of pages are not subsequently referenced at all by the migrated process and so never need to be moved to the destination. The main disadvantage is the overhead for the process after migration to re-establish a working set on the destination host. In addition, the continued storage of pages on the source host is a form of residual dependency. Sprite [8] uses a variation of this approach, with dirty pages on the source being flushed to a file server, from which the destination fetches data as page faults occur.

At this stage we have limited ourselves to a straightforward implementation of memory transfer. Since Amoeba is not aimed at hard real-time applications, latency is not an over-riding concern, especially not at the cost of the increased total cost caused by pre-copying. Implementation of lazy copying is not a sensible choice either, since Amoeba does not support virtual memory facilities, to avoid the performance penalty. Before deciding whether to implement lazy copying in Amoeba it would be necessary to look at the total system performance effect of implementing both virtual memory and lazy copying.

5 Design details of the Amoeba mechanism

5.1 State transfer

The complete state of a process in Amoeba includes information about that process held in the kernel, in addition to the process's user space. The latter simply consists of the contents of the process' memory segments. Kernel information includes memory mapping for each segment, machine registers for each thread, communication state for each thread, and inter-thread synchronisation information.

In general, each thread of a process may be executing (or ready to execute) in user space, executing in the kernel (as a result of a system call), or blocked. Copying the state of a process with threads that are all either blocked or executing/ready in user space is essentially straightforward. However, copying the state of a process with a thread executing in kernel state is anything but straightforward. The reason is essentially that the kernel is not itself being migrated, and so to encode the state of kernel execution in such a way that this state can be identically reproduced in the kernel of the destination machine is a difficult problem. Fortunately, it is also unnecessary. Execution times in the Amoeba kernel are always short, and terminate in either a return to user space, or in the thread becoming blocked. It is satisfactory therefore, when freezing a process for migration, to wait until it does not have a thread executing in the kernel. In other words, system calls are allowed to run to completion except when they cause the thread to be blocked.

There is one additional constraint applied to the state of a process before allowing it to be frozen. If a thread is actively transferring data (sending or receiving a request or reply message), then this is allowed to complete before freezing the process, despite the fact that the thread may be blocked for short periods of time during the transfer. This avoids timeouts in the RPC protocol between communicating processes.

Capturing the state of a thread reduces now to two cases - either the thread was executing in user space before the process was frozen, or it was blocked. In the former case, the state consists of the relevant parts of the process's user memory (in particular the thread's stack) plus its kernel state. In the case that a thread is blocked, then its kernel state in addition contains the reason it is blocked. There are only two reasons that an Amoeba process may be blocked - it is waiting to receive a message (request or reply), or it is waiting for a mutex to be unlocked. Each of these reasons is readily encoded, including its parameters (eg address of mutex).

5.2 Communication with migrating process

In accordance with the objective of transparency, neither a process being migrated, nor processes communicating with it, should be aware of the migration in order to continue to communicate successfully. However it is acceptable for there to be a communication delay caused by the migration - Amoeba is not a real-time operating system, and process migration is one of a number of causes which may result in a greater-than-normal communication time. It should also be noted that each thread of a process may be engaged in two communication transactions at one time - this occurs when the thread, in processing a received request, sends a request to another process.

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 with a process whose migration has completed: As indicated, the objective is for the migrated process to have no residual dependency on the processor from which the process was migrated. This means that communication with the process should occur after migration in the same way as before, without the use of the original processor to re-route messages. Essentially, this is handled by the fact that FLIP network addresses are location-independent. It is simply a matter of ensuring that the addresses used by the process are migrated with it.

Communication during migration: Process migration 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. These messages can be dealt with only after completion of the migration. There are at least two ways of dealing with them:

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

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 “migration pending” status response to indicate that the process is temporarily unavailable to receive messages. This is not regarded as an error condition, and the sender is expected to handle this case by trying again later. Because this 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 to the FLIP protocol of [24]. Note that 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 in fact no need to handle this case specially - the mechanisms in the previous sections work equally well when the process resumes execution without having migrated. This is a consequence of the fact that “locate” protocols are used to find the process after migration.

5.3 Control

It was decided to implement the control of the process migration mechanism by providing a *migration server* in each host processor. The decision to implement the mechanism in a distinct server minimises the additional logic required in the kernel and the process server. It was also necessary to decide whether there would be a single central migration server, or multiple servers. A single server would have the advantage of using resources in only one host processor, and of being a single point to which all migration requests are directed. However, it also has the disadvantage of being a potential bottleneck when several migration requests are active simultaneously, and of incurring a higher communication overhead, especially when the single migration server is not running on either the source or destination processors. Therefore the approach of a migration server in each host processor was adopted.

The migration server in the source and destination hosts cooperate to achieve the migration of a process. Migration is initiated by sending a request to the migration server in the source host. This server contacts the migration server in the destination host, which then carries out the majority of the work.

The sequence of events for migration of a process is as follows:

1. The migration server on the source host receives a request to migrate a specified process to a destination host.
2. The source migration server freezes the process by sending a “suspend” request to the process server on the source host. Any messages received for the process after this point cause a “process is migrating” response, generated by the FLIP layer on the source host. These are recognised by FLIP on the sending host, which will retry after a suitable delay.
3. The source process server sends a “process descriptor” for the source process to the migration server on the destination. This contains the state of the process, including all state information held by the kernel, and the size

and number of memory segments. The destination migration server uses this to set up a copy of the process on the destination host, by making requests to the destination process server to set up the kernel state and to allocate memory for the process segments.

4. The destination migration server sends a series of RPC requests to the source process server, copying the segments from source to destination. The RPC replies which carry the segment contents are transmitted directly from source segment to destination segment without memory-memory copying - this is important for performance reasons.

5. The migration is completed by the passing of an *execution token* for the process from the source to the destination migration server in a message exchange between the two. When the source migration server has sent the token, it removes the process on the source host. When the destination migration server receives the token, it sends a “process restart” request to the destination process server to indicate that the process is executable.

6. Any messages sent to the process on the source will now receive a “not here” reply, causing the FLIP layer on the requesting host to use its locate mechanism to find the process on the destination host.

5.4 Errors During Migration

Various errors can occur to prevent successful completion of a migration. These include failure of the source or destination host, an unrecoverable communication error, and insufficient memory for the process’s segments on the destination host. Under these circumstances it is desirable that the process be removed from the destination and resume normal execution on the source host. The design presented here achieves this, except for the (inevitable) loss of the process if the source host fails before migration has completed.

The passing of the execution token eliminates the possibility that a migrated process can be executing on *both* source and destination hosts simultaneously. Only the holder of the token is free to resume its copy of the process. An unrecovered communication error can cause loss of the token and therefore of the process, but not duplication. The destination uses a timeout to ensure that it will not retain a migrating process indefinitely waiting for the execution token.

6 Current status and future work

The work described in this paper is partly implemented - we have a prototype version of the migration server which successfully migrates processes. The prototype does not handle the transfer of communication state, nor the generation and handling of “migration pending” status responses, and so migration is not at this point completely transparent. We are therefore not yet able to study its effectiveness as a general load-balancing tool, although the prototype is being used for experimental studies applied to test processes which do not require communication transparency.

After complete implementation of the design as described in this paper, possible future work includes:

• Extension of the mechanism to cater for the migration of processes engaged in Amoeba group communication - this has not been included at this stage because we have not had the time to investigate it;

• The use or adaptation of process migration for fault tolerance.

7 Acknowledgments

We are grateful for a research grant from the Institute of Computer Systems Engineering and Assurance, University of South Australia, which made this work possible, and for a University of South Australia internal research grant which contributed to the purchase of equipment.

Undertaking redevelopment work on an operating system, when the original development team is located halfway across the world, is arguably a foolhardy exercise. It has been made possible only by the help received from Andrew Tanenbaum and the Amoeba project, first in acquiring Amoeba and then in providing support and information. Frans Kaashoek kindly provided the unpublished [27] which provided some of the ideas for our work. Greg Sharp and Kees Verstoep deserve special thanks for their patient assistance and unfailingly prompt responses to our questions.

Thanks are due also to the referees, whose feedback has helped to improve the quality of the paper significantly.

Earlier work by Michael Carrucan and Aart van Halteren in the context of student projects provided a very useful starting point for this project.

8 References

[1] 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.

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

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

[4] W. Zhu, “The Development of an Environment to Study Load Balancing Algorithms, Process Migration and Load Data Collection”. PhD thesis, University of New South Wales, 1992.
[5] S. Zhou, “A Trace-Driven Simulation Study of Dynamic Load Balancing”. *IEEE Trans. on Software Eng.* vol. 14, no. 9, 1988.
[6] Y.-T. Wang and R.J.T. Morris, “Load Sharing in Distributed Systems”. *IEEE Transactions on Computers*. vol. C-34, no. 3, pp. 204-217, 1985.
[7] A.S. Tanenbaum, *et al.*, “Experiences with the Amoeba Distributed Operating System”. *Communications of the ACM*. vol. 33, no. 12, 1990.
[8] 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.
[9] A. Barak and A. Shiloh, “A Distributed Load-balancing Policy for a Multicomputer”. *Software - Practice and Experience*. vol. 15, no. 9, pp. 901-913, 1985.
[10] M.L. Powell and B.P. Miller, “Process Migration in DEMOS/MP”, in *Proc. 9th Symposium on Operating System Principles*. pp. 110-119, 1983.
[11] G.J. Popek and B.J. Walker (eds.), *The LOCUS Distributed System Architecture*, *Computer Systems Series*, Cambridge, Mass.: MIT Press, 1985.
[12] Y. Artsy and R. Finkel, “Designing a Process Migration Facility: the Charlotte Experience”. *Computer*. vol. 22, no. 9, pp. 47-56, 1989.
[13] 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.
[14] E. Zayas, “Attacking the Process Migration Bottleneck”, in *Proc. 11th ACM Symposium on Operating Systems Principles*. Austin, TX, ACM, pp. 13-22, 1987.
[15] J.M. Smith, “A Survey of Process Migration Mechanisms”. *ACM Operating System Review*. vol. 22, no. 3, pp. 28-40, 1988.
[16] M.R. Eskicioglu, “Process Migration in Distributed Systems: A Comparative Survey”. Technical Report TR 90-3, University of Alberta, 1990.
[17] 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.
[18] 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.
[19] 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.
[20] A.S. Tanenbaum and R. van Renesse, “Distributed Operating Systems”. *Computing Surveys*. vol. 17, no. 4, pp. 419-470, 1985.
[21] 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.
[22] R. van Renesse, “The Functional Processing Model”. PhD thesis, Vrije Universiteit, Amsterdam, 1989.
[23] A.D. Birrell and B.J. Nelson, “Implementing Remote Procedure Calls”. *ACM Transactions on Computer Systems.* vol. 2, no. 1, pp. 39-59, 1984.
[24] 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.
[25] “Amoeba Reference Manual”. Vrije Universiteit and Stichting Mathematisch Centrum, 1992.
[26] S.J. Mullender, “Process Management in a Distributed Operating System”. Technical Report CS-R8700, Centre for Mathematics and Computer Science, Amsterdam, 1987.
[27] F. Douglis, M.F. Kaashoek and G.J. Sharp, “Amoeba 6.0 Kernel Interface Specification”. Unpublished draft Vrije Universiteit, 1992.
[28] Y. Hollander and G.M. Silberman, “A Mechanism for the Migration of Tasks in Heterogeneous Distributed Processing Systems”, in *Parallel Processing and Applications*, Chiricozzi, E. and d’Amico, A., (eds.), North-Holland, 1988.