Processes Migrate in Charlotte

by

Yeshayahu Artsy
Hung-Yang Chang
Raphael Finkel

Computer Sciences Technical Report #655
August 1986

Processes Migrate in Charlotte*

Yeshayahu Artsy
Hung-Yang Chang
Raphael Finkel

University of Wisconsin – Madison
Computer Sciences Department

August 1986

Abstract

Process migration in a distributed operating system is a facility to dynamically relocate processes among component computers. In recent years, several studies have been conducted concerning the need for process migration and the algorithms to perform it efficiently. Only a few successful implementations of process migration have been reported.

A process-migration facility has been implemented in Charlotte, a message-based distributed operating system. Process migration policy is decided by a user-level utility called the *Starter*, while the mechanism is handled by the kernel. A distributed squad of Starters can enforce regional and global process migration policy. Several migration efforts can proceed in the system concurrently. Migration can be aborted due to a change in the load; the migrating process can be rescued in many cases of machine failure. Migration is transparent to the migrating process and to other processes communicating with it. Once a process is migrated out of a machine, no trail of stubs or dangling links is left to interfere with future communication.

This paper gives an overview of Charlotte, discusses the design of the process migration facility, details its implementation, and gives some preliminary cost results. Process migration was implemented in mid-1985 and has been used experimentally since then.

*This work was supported in part by NSF grant MCS-8105904 and by DARPA contracts N00014-82-C-2087 and N00014-85-K-0788.

CONTENTS

1. Introduction

In a distributed operating system, distribution of processes to processors has two different but related motivations: to improve parallelism of applications and to share load among processors. An application programmed for parallel processing wishes to distribute its cooperating processes in order to maximize their parallelism and minimize communication delay. Load sharing among processors tries to distribute the processes in order to reduce the "wait-when-idle" situation, in which processes wait in one processor while another processor is idle. There are other motivations as well, such as to bring a process to the processor that has a special device or to move an urgent process to a lightly loaded processor.

Distribution of processes can be achieved through initial placement, which is relatively cheap due to the simple state transfer involved [1]. However, process placement cannot fully cope with the dynamic nature of distributed systems, as loads and service needs may fluctuate in unforeseen ways. **Process migration** means to transfer a process, at any time in its life cycle, to another processor. Process migration lends itself to dynamic resource allocation in distributed systems.

There are many complexities involved in the mechanism of process migration. A process must be *detached* from its current environment, *transferred* with all its relevant context information, and *reinstalled* in the new environment. This facility requires remapping location-dependent values in the process context, redirection of communication channels, and handling incoming messages while the process is moving. Moreover, such a facility must not affect the behavior of the migrating process or other processes at all. It is interesting to notice, therefore, that only a few implementations of process migration have been reported [2,3,4,5]. Charlotte [6] offers a uniform, location-transparent message-based communication facility. In this environment, process migration mainly requires readdressing the process's communication links. However, Charlotte also offers a powerful and flexible IPC, so process migration has to cope with many complexities that may arise due to various communication scenarios during the transfer of a process.

A different class of problems arises in the realm of migration policy. A process migration policy has to dictate *when* to migrate *which* process to *where*, based on some notion of profitability. Many policies have been proposed and studied using analytic and simulation tools [7,8,9,10,11,12]. Process migration in Charlotte was designed as a real-system tool to evaluate such algorithms. In this paper we discuss mainly our solution to the mechanism issues and how our facility supports migration policy dictated by

higher-level facilities.

The motivation for incorporating a process migration facility in Charlotte is threefold:

- to provide a testbed for the study of load sharing policies and migration algorithms in a distributed operating system.
- to enhance Charlotte's function as a testbed for research in distributed algorithms [13].
- to demonstrate the feasibility of an efficient implementation in the context of a powerful but complex IPC mechanism.

Our main design goal was to embed the process migration facility into Charlotte without changing its major features and structure and without affecting the behavior of processes and users.

As a result, the Charlotte process migration facility has several important aspects. First, it clearly separates the decision-making (policy) facility from the migration mechanism. The former is relegated to a user-level utility process, while the latter is embedded in the kernel. This separation allows using and testing different policies without modifying the kernel. Secondly, migration is transparent to both the migrating process and the processes connected to it. While a process is migrating, other processes can continue to communicate with it (although the communication will suffer a small delay) or even destroy or move their connections with it. The migrating process sees the same execution environment before and after migration. Thirdly, migration can be aborted in the middle; the migrating process can even be rescued under certain circumstances when either the source or the destination machine crashes (for instance, if the destination machine crashes before the source machine has committed itself to the migration).

This paper is organized as follows. The next section overviews Charlotte, its major components and IPC mechanism. In Section 3 we discuss design issues of our process migration facility, and in Section 4 we detail its implementation. Section 5 presents an evaluation of execution-time and code costs. Some conclusions are drawn and future directions are discussed in Section 6.

2. Charlotte environment

Charlotte is a distributed operating system for the loosely-coupled Crystal multicomputer [14]. The current configuration of Crystal consists of 20 VAX-11/750 machines connected by an 80 Mbps Pronet token ring. Charlotte was designed (1) to explore operating system design that provides an inexpensive but

rich environment, and (2) to serve as a testbed for research in distributed algorithms. We summarize here the major components of Charlotte, its IPC mechanism and its implementation. More detailed discussion has appeared elsewhere [6, 15, 16].

2.1. System Components

Charlotte consists of a kernel that runs on top of a communication package called the *nugget* [17] and a battery of user-level *utility processes*. An identical copy of the kernel executes on each machine. The kernel supports *processes* and communication *links*, implements the interprocess communication (IPC) and process migration mechanisms, and provides a round-robin short-term scheduling. Charlotte does not support paging.

The nugget provides reliable transfer of variable-length messages (limited to 2K bytes). The interface to the nugget is via procedure calls. The nugget interrupts its client upon message arrival and transmission completion by invoking a handler. Charlotte allows only the kernel to access the nugget.

Utility processes are distributed throughout the network. They cooperate to allocate resources, provide higher-level services such as file and connection services, and enforce control policies such as medium to long-term scheduling, process placement and migration. For reasons of efficiency and reliability, a service can be provided by a *squad* of processes that implement one utility; members of a squad all run copies of the same code but divide responsibilities and load among themselves. Processes (including utilities) communicate with each other by exchanging messages. The location of a given utility, or any member of its squad, is transparent to all its clients. Processes communicate with the kernel by kernel calls, some which are restricted to special utilities.

Process migration is performed through the cooperation of the KernJob and Starter utilities. The *KernJob* is the only utility that runs on each machine. It provides a path between its kernel and processes that need control over that kernel's process. The *Starter* manages memory allocation, process creation, periodic (medium to long-term) scheduling, and process migration. Members of the Starter squad receive periodic load information from the KernJobs of each of the machines they control. Other utilities such as the *File Server*, *Switchboard*, and *Command Interpreter* perform services with no direct relationship to process migration.

2.2. Interprocess Communication

Processes communicate by exchanging messages over links. A *link* is a logical, full-duplex connection between two processes, each of which has a capability to one end of the link. Both processes connected by a link have the same *rights* over it: to send and receive messages, to cancel a request to send or receive a message, to move the link to another process, or to destroy the link. Both processes can do any of these operations concurrently.

Processes never refer to each other directly, but only through the links that connect them. The process identifier and machine location of the other end of a link are invisible to the process, although the kernel maintains full, up-to-date information about the links held by processes it controls: their logical address[1] and states.

The Charlotte IPC mechanism is a descendent of Arachne [18], which itself employs a distributed IPC mechanism based on Demos [19]. Charlotte extends this IPC in several ways. Process-level communication is:

- *non-blocking*: A process can continue executing after posting a send and/or receive request on any or all of its links. The process can later interrogate the result of these requests.
- *unbuffered*: The kernel transmits a message from the sender's address space to the receiver's space without intermediate buffering. For efficiency reasons, the kernel maintains a small cache of messages received from remote senders before the matching local receive has been posted. Processes are oblivious to the existence of this cache and to the way cache misses are handled.
- of *unlimited size*: A process can send or receive a message of any size, from zero to its entire address space. Both the receiver and the sender are notified when a message overflows the receiver's buffer.

In addition, the Demos link concept was modified so that

---

[1] A logical address in Charlotte is the 3-tuple: machine number (1 to n, where n is the number of machines in the system), process identifier, and link number.

- links are *duplex*, in the sense that both processes can perform all link operations concurrently, as explained above, and
- all interprocess links are identical: no special-case links (such as *reply* or *request* links) are supported.

Here is a summary of the Charlotte IPC primitives.

MakeLink (var end1, end2 : link)

Create a link and return references to its ends. (The creator then can transfer one end to a client or a server or transfer both ends to two processes.)

DestroyLink (myend : link)

Destroy the link at one end. The process at the other end is notified of this event. That process can still obtain the result of previously posted communication requests, but cannot post new ones. The other end is disposed only when that process explicitly calls DestroyLink.

Send (L : link; buffer : address; length : integer; enclosure : link)

Post a send operation on a given link end, optionally enclosing another link end.

Receive (L : link; buffer : address; length : integer)

Post a receive operation on a given link end. L can be a specific link or *AnyLink*.

Cancel (L : link; d : direction)

Attempt to cancel a previously-started *Send* or *Receive* operation. L can be *AnyLink*.

Wait (L : link; d : direction; var e : description)

Wait for an operation to complete. L can be a specific link or *AnyLink*. The direction can be *Sent*, *Received*, or either. The description returns the success or failure of the awaited operation, its link, direction, number of bytes transferred, and the enclosed link (if any).[2]

GetResult (L : link; d : direction; var e : description)

Ask for the same information returned by *Wait*, but do not block if the operation has not com-

[2] *SendWait* and *ReceiveWait* primitives, which combine *Send* and *Receive* with *Wait*, are also supported.

pleted. GetResult is a polling mechanism.

2.3. Kernel Design and Implementation

The kernel is structured into three modules, each of which executes a Modula process [20]. (We refer to these processes as *tasks*, to distinguish them from Charlotte user-level processes.) The *envelope* task performs kernel calls or dispatches them to other kernel tasks; when all kernel tasks are idle, the envelope chooses a user process to run until its CPU quantum expires, it gets blocked, or an interrupt occurs. The *fsa* task implements the IPC facility, including a protocol for flow control and error recovery. The *communication* tasks manage packet delivery from and to the nugget layer and the fsa task. Kernel tasks communicate with each other by means of queues of work requests, implemented by the atomic queuing instructions of the VAX.

The flexibility and power of our IPC have a price: complexity. The number of possible scenarios per each link is enormous, taking into account all (possibly concurrent) process operations and kernel actions. Some of these scenarios are very complicated. (Consider for example the case where both ends of a link send messages to each other; before they both receive each other's message, one process tries to cancel its message and to move the link to another process, while the other process tries to destroy the link.) Designing a protocol that can handle all the cases efficiently was a difficult task indeed.

The fsa task drives the protocol, using a finite state decision table. Each link end has a state, which includes the state of its send and receive 'ports'. Given the current state and an input request, which is either a kernel call or an inter-kernel message, the next state and action are decided. The action may include a message to a remote kernel. To reduce the huge number of states, and to overcome the sparseness of the table, the fsa is decomposed into four conceptually independent automatons, each handling a different function: Send, Receive, Destroy, and Move. The details of the protocol and its complexity are elaborated elsewhere [15]

3. Process Migration: Design decisions

The design of process migration in Charlotte is based on the following principles.

Policy-Mechanism separation

In order to be able to test and evaluate different policies, migration policy is outside the kernel in user-level utility processes. Since the Starter already controls process creation and memory allocation, it was the natural candidate to manage process placement and migration. The Starter was modularized so that the migration policy module can be easily replaced by various researchers. The Starter is structured to let each Starter process decide regional policy in the cluster of machines it controls and coordinate with its peers to achieve global policy.

The kernel performs the mechanism of process migration. It accepts policy decisions from the Starter through a simple kernel call interface. The kernel transfers the address space of a migrating process and its associated kernel data structures, marshalling and remapping them as necessary. If a requested migration cannot be completed, for instance due to process termination or machine failure, the kernel notifies the Starter appropriately. The kernel does not automatically initiate or refuse a migration by itself.

The kernel also provides statistical data about its internal resources, CPU and communication loads, process and link activities. The statistics collected are as broad as possible, to support all possible parameters in the Starter's migration algorithm.

Mechanism-Mechanism independence

The mechanisms of both statistics collection and process migration are designed not to interfere with other kernel mechanisms, in particular the IPC mechanism. Some interaction between the mechanisms does exist. For instance, a process might be unable to migrate in some circumstances because of its communication state. Also, process migration can take advantage of existing facilities of the IPC mechanism. However, the migration protocol does not intermingle with the IPC protocol. Thus, each mechanism is simpler and can be modified independently.

Transparency

Since location transparency is one of the principles of Charlotte, it was natural to extend this principle to process migration as well. Neither the user nor any process, including the migrating process, is aware of the fact of migration (except for a short delay in response). In particular, processes connected to the migrating process are not affected at all. They can send messages across the links, or even move or

destroy them, while the process and its links are relocated; in fact, their kernels see the links connected to the migrating process as simply being moved, as they are when moved by a *Send* call.

In addition, we want the migrating process to be minimally affected. Hence it is 'frozen', that is, its activity is stopped, only for the shortest time possible.

Reliability issues

Process migration in Charlotte is reliable and can survive any single machine crash (and in many cases multiple machine crashes). Since the kernel uses the nugget's reliable message delivery, migration is reliable 'by default' when no crash occurs. When the source machine or the destination machine crashes, the surviving machine will abort the migration. However, migration is not totally fault tolerant, in the sense that the migrating process cannot continue to run in all cases. If one of the two machines involved in migration crashes while the process's state is temporarily inconsistent, then it is simpler to destroy the process rather than to restore its state. When a machine that has the remote end of a link held by the migrating process crashes, migration completes without any recovery effort.

Concurrent, multiple migration efforts

Since each Charlotte machine is autonomous, migration should concern only the two machines directly involved. Thus, any two machines can be engaged in a migration effort, independently of other machines or even other migration efforts between the same machines.

There are two problems to overcome in dealing with multiple migrations. First, when several processes are transferred to a single machine, the machine needs to have sufficient resources for all these processes. Therefore, during the migration negotiation phase (see below), a machine reserves the memory and kernel data structures needed for the migrating process before agreeing to accept it. Second, when two processes connected by a link migrate concurrently, each kernel might have a wrong conception as to where the other end of the link resides. However, this case is equivalent to the case when two processes concurrently move a link they share, and we can employ a similar protocol to the one used in that case [15]. Another issue of concern in multiple migrations is to avoid flooding a lightly loaded machine or draining a heavily loaded machine. This issue belongs to the migration policy, and is left entirely to the Starter.

No stub left behind

A process is transferred entirely along with all its kernel data structures. We do not want to leave any process or link skeleton in the source machine to handle future communication directed to the process. Since every kernel maintains absolute information about the links of its controlled processes (as opposed to hints), all kernels maintaining the remote link ends of a migrating process are told the new addresses before the migration can complete. Likewise, all incomplete transactions initiated by or directed to the migrating process are transferred to the destination machine for completion.

4. Implementation Details

The process migration facility was fit into the production Charlotte kernel as three independent modules: one to collect the necessary statistics, another to interface with utilities, the third to perform the process transfer. The first module executes as a separate task that awakes periodically to perform sampling; the others are invoked by the fsa and the communication tasks (and thus are synchronized with IPC-related events). We describe each of them in turn.

4.1. Statistics Gathering

Statistics are collected in three different modes:

(a) *event sampling*, in which events such as message completion or process termination are instrumented to record their statistics.

(b) *interval sampling*, in which a sample is taken of the current CPU load (expressed in the number of ready-to-run processes), and the network load (expressed in the number of messages awaiting transmission) at fixed time intervals.

(c) *periodic statistics*, in which the average of several interval samples is calculated at a larger time interval.

The kernel collects and delivers the following statistics:

- *machine load*, consisting of the number of processes and links (whether currently active or not), the average CPU load, and the average network load.

- *process resource usage*, consisting mainly of the average and total CPU and network utilization of each process, its state, and its total communication (expressed in number of packets) to local and remote processes.
- *link statistics*, for the most active links per process, giving totals of the number of packets sent and received over each link.

To simplify the transfer of statistics, the kernel collects them in a memory area provided by the KernJob. In this way, the kernel needs only to signal the KernJob at the end of a period, but no data need to be transferred. The KernJob then transfers the statistics to the Starter as a message. The intermediation of the KernJob is necessary, since the Starter squad is distributed throughout the network.

Members of the Starter squad exchange information about the load of the machines they control. For example, we have implemented Barak's *gossiping* algorithm [21], which avoids broadcast and quickly calculates the estimated global load. Each Starter process records the loads of the machines it controls as well as recently-received load information from other Starters. Each Starter process derives a new estimate of the current system load. It then sends to one of its peers (selected randomly) its new estimate, its load information and the most recent information received from others. Barak has shown that in the case where information is exchanged between kernels that control a single machine a piece, each kernel can know the load of every machine in the system, and can derive an estimate that is very near to the 'real' average load, both in a very short time.

4.2. Kernel-Utilities Interface

Four new kernel calls were added.

HandShake (what : action_type, buffer : address)

called by the KernJob to request the kernel to start or stop collecting statistics.

MigrateOut (who : process, where_to : machine)

called by a Starter process, either directly, if it resides in the source machine, or otherwise indirectly through the KernJob of that machine.

MigrateIn (which : (process, machine), Accept : boolean, memory_blocks : list of (address, size))

called by a Starter process either directly or via the KernJob, as before. Its purpose is to let the

Starter that controls the destination machine approve or disapprove a migration request that has been announced to the Starter by the kernel of the destination machine, or to preapprove a migration attempt after negotiating it directly with the Starter that controls the source machine. The tuple (process, machine) refers to the source machine. If the Starter agrees to accept a process, the Starter furnishes a list of memory blocks, one for each of the process's segments (whose size is known from the migration request);[3] the process's image will be copied into these blocks.

CancelMigration (which : (process, machine))

tries to retroactively terminate a MigrateOut or a MigrateIn request. It fails if the kernel has already committed the migration (even if it has not yet been completed).[4]

The return code of each of these kernel calls specifies only whether the requested operation can be started. When it gets around to it, the kernel notifies the appropriate Starter, via the KernJob, that statistics have been collected, a migration offer has been received, or that a migration has been completed or aborted. The notification mechanism appears as a special communication link interface to the KernJob. The kernel puts an appropriate message in an imaginary link owned by the KernJob; when the latter does a Receive from *AnyLink*, this message is received before any other pending message.[5]

4.3. Migration Protocol Overview

After a process is selected by a Starter to migrate out, migration is performed by the kernel in three phases:

(a) negotiation phase, in which the kernel of the source machine makes a migration offer to the kernel of the destination machine, and from then to the Starter that controls this machine.

(b) transfer phase, in which relevant kernel data structures are assembled and transferred together with the address space of the process.

---

[3] In the VAX implementation, each process has two segments.

[4] CancelMigration has not been implemented, and MigrateIn is used only to approve or disapprove a previously received migration offer. However, the protocol supports both features; they will be put in the kernel soon.

[5] The kernel scheduling algorithm has also been modified to run the KernJob and Starter preferentially over other processes.

(c) cleanup phase, in which the process's state is completely removed from the source machine, and the new image is installed and activated in the destination machine.

Phase 1: Negotiation

The migration offer prepared by the kernel specifies the size of the process's segments, its link table and its current active links. It includes also some information about the process's CPU and network utilization (totals and averages) and the process's age.

If the Starter has preapproved this migration (that is, called MigrateIn in advance) and the kernel has the necessary resources for the new process, then the kernel replies with a *migration accepted* message. Otherwise, the kernel delivers the details of the offer to the destination Starter, via the KernJob. Acceptance or refusal is returned to the source kernel.[6]

Upon acceptance, the kernel also reserves the necessary resources and forks an internal task to accept the process's image across the network. If migration is refused, both kernels clean up their migration-related state, and the Starter processes of both ends of the migration are notified.

The migration-accepted message also serves as a *commit point* for both kernels. Once sent, the destination kernel cannot cancel the migration. Once starting the second phase, the source kernel cannot cancel it either. Until then, it can send a *migration regret* message to the other kernel, for instance due to an abrupt termination of the the migrant process, or a sudden decrease in its load. In such a case, each kernel cleans up state information relevant to the migration and notifies the Starter of the migration abortion.

Machine failure before the commit point has minimal affect on migration. If one of the two machines crashes, the other cleans up its state. The migrating process either is not transferred (if the destination machine crashes), or dies (if the other machine crashes). If another machine crashes, only the process's links that point to that machine need to be destroyed.

---

[6] In the current version, only one migration effort at a time per machine is supported. Hence, if the polled machine is already involved in a migration rendezvous (even in evaluating a former migration offer from another machine), the kernel refuses the offer right away.

4.3.1. Phase 2: Transfer

At this phase the migrating process is 'frozen', in the sense that its activity is suspended.[7] This phase consists of three steps, executing partly in parallel.

(a) *Image transfer.* The kernel of the source machine transfers the image of the process in as many physical packets as needed as fast as the other machine can receive them. These packets are copied by the network controller on the other machine directly to the space reserved earlier by the kernel.

(b) *Link Update.* This step imitates the link moving for any of the migrating process's links. The kernel of each of these link ends is told the new logical address of the link. Each of these kernels acknowledges this message. Should any of the machines fail at this point, the source kernel simulates an acknowledgement followed by a link destruction request for all the links pointing to that machine. Once all acknowledgements are collected, the source kernel can safely transfer all the process's data structures, since no more messages will be received across its links.[8]

Until this point, all messages received for the migrating process at the source machine are buffered by the kernel. The headers are transferred with the process at the next step. The data are not transferred; the other kernel simulates a kernel-buffer cache miss when the migrating process is ready to receive them. From this point, the destination kernel might receive messages for the migrating process, even before the process's image and state are fully received. Hence, it buffers them until the next phase.

During this step the link states of the process are temporarily inconsistent. Some of the kernels already know the new addresses, some don't; some of them might have sent messages to the source machine, others to the destination machine. More important, the two kernels involved in the migration have partial information of the link states of the migrating process. Hence, should either the

[7] The process is frozen during the former phase should it create or receive new links. Such a link modification request is buffered until migration completes. Destroying links or giving them away does not require that the process be frozen immediately.

[8] We considered an alternative in which the destination kernel tells all other kernels the new addresses, in order to reduce the source kernel's work. However, this method would delay the source kernel in transferring the process until it can be sure that no more messages will be received across its links.

destination machine or the source node crash, the other machine must kill the migrating process. We prefer this option over negotiating retransfer of possibly lost messages and link address updates, because it is simpler and, in any case, the situation is very unlikely to occur.

(c) *State transfer.* All kernel data structures relevant to the process, in particular its process descriptor, link descriptors, descriptors of pending or completed events, and received messages, are packaged in as many physical packets as needed and transferred. The source kernel marshals only the necessary data items; in particular, it dereferences pointers that use kernel virtual or physical addresses. The other kernel reinstalls those pointers appropriately upon receipt.

4.3.2. Phase 3: Clean-up

This phase is short and simple. When all packets have been sent, the source kernel deletes all data structures relevant to the migrating process and notifies its Starter. When all packets have been received, copied and/or marshaled back, the destination kernel adds the new process to the appropriate (ready or waiting) list. All buffered messages are reactivated, starting with those received by the source kernel. The Starter is told that migration has completed.

4.3.3. Final comments

We have touched on situations when a machine crashes during the various phases of migration, and shown an intuitive solution for each case. Recovery from a crash (with respect to the migration) always puts the machine back to a consistent state, although sometimes the process must be terminated.

Our protocol correctly handles complex situations, such as when a process migrates concurrently with other processes linked to it. The problem in such a case is whom to tell the new link addresses, or rather how to guarantee that all the destination kernels know the correct link addresses when each migration is finished.

We have mentioned that the protocol handles messages received during the migration. But what about link motion or destruction during that time? Until the second and during the third phases, such requests are delivered to the lower-level communication protocol; during the second phase, such requests are buffered (that is, delayed until the third phase). Since migration is guaranteed to complete within a certain (short) time, delaying these requests cannot introduce a risk of deadlock or error.

As mentioned before, policy is encapsulated within one module of the Starter. This module interfaces with other modules via a single procedure called at every event reported by the kernel that returns a policy decision. Researchers who want to investigate different process migration policies need only code this module and link it to the Starter.

5. Cost of Migration

We present here some initial measurements of the migration facility. The result of this measurement should be viewed from the perspective of Charlotte environment. Charlotte resides on top of the nugget, which incurs an interprocessor reliable message delivery delay of 11 ms for each 2k byte packet. Charlotte itself is layered (kernel and utilities), and the kernel itself is multi-tasked with a task switch overhead of 0.4 ms. A Charlotte message incurs an overhead of 23 ms for a pair of send and a receive operations on two processes on two machines. It takes 10 ms if both sending and receiving processes are on the same machine.

The following are results of about 100 migrations (for each test) performed in a two-node network:

(1) It takes, on average, 323±11 ms to migrate a "simple" process (a linkless process of about 32 KB image size) between two nodes. This time includes the negotiation phase.

(2) Each additional 2 KB (that is, one communication packet) of image size adds 12.2 ms to the average migration time.

(3) Each additional link held by the migrating process adds insignificantly to the elapsed time and to the kernel overhead if the remote link end resides in the source node, and 9.9 ms otherwise.

(4) The kernel spends, on average, 125±12 ms migrating a process out, and 48±5 ms migrating one in.

(5) Without the migration facility, the kernel code size is 120 KB and kernel data size is 90 KB. This includes kernel tracing and debugging code and run-time facilities that support debugging. The migration facility adds another 40 KB to the code and 20 KB to the data. An additional facility for tracing the migration protocol (as well as to collect some of these results) requires 12 KB.

(6) The kernel time overhead in collecting statistics is about 1% of overall elapsed time.

6. Conclusion

Charlotte is intended to be used as a testbed for distributed applications. Process migration complements the initial placement facility of Charlotte. The combination makes Charlotte a flexible execution environment with full fledged dynamic resource allocation capability.

The migration facility is itself a research vehicle to study load sharing policies for distributed applications. The design of the migration facility meets the above goal by separating policy from mechanism and by providing an inter-machine statistics-gathering facility.

The policy and mechanism of process migration are implemented at different levels of the operating system. The policy is encoded in a user level utility process while the mechanism is embedded in the kernel. This design allows an easy replacement of process migration policies without modifying the kernel. Also, the kernel implementation of the mechanism ensures high performance of such operations.

The Charlotte kernel maintains a rich set of statistics on process and machine activities. The statistics include per-process and per-machine resource utilizations sampled frequently, an IPC event summary, and medium-term load averages. The statistics are supplied by each kernel to its policy process via the standard message interface. The policy process can reside on any remote machine.

Our implementation of the migration mechanism is straightforward in most cases but tricky in the re-routing of links for the migrating process. This is because re-routing interacts with the IPC protocol, which contains abundant complex cases due to the asynchronous, cancellable, and connection-based nature of the IPC. Our implementation circumvents the problem by adding the migration protocol in a way that requires no modification of the original IPC protocol. Both concerned kernels of a migrating process temporarily buffer both control and data messages received for the links of the migrating process. The source kernel then simulates the link-moving paradigm to tell the holding kernels of remote link ends the new link address. And then, those buffered control and data messages are fed through the IPC mechanism in the destination kernel to restore communication.

Projects are underway investigating the use of the process migration and process placement facilities of Charlotte, to enchance the performance of distributed applications.

7. Acknowledgements

Process migration in Charlotte was inspired by our discussions with Amnon Barak from the Hebrew University of Jerusalem. We owe our gratitude to Cui-Qing Yang for modifying the Charlotte utilities to support process migration. Miron Livny has also contributed some valuable suggestions. We would like to thank other members of the Charlotte group for helpful comments, especially Marvin Solomon and Michael L. Scott.

8. References

1. D. Eager and E. D. Lazowska, "Dynamic Load Sharing in Homogeneous Distributed Systems," Technical Report 84-10-01, University of Washington (October 1984).
2. A. B. Barak and A. Litman, "MOS: A Multicomputer Distributed Operating System," *Software — Practice and Experience* **15**(9) pp. 901-913 (September 1985).
3. M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proc. of the Tenth Symposium on Operating Systems Principles*, pp. 2-12 (December 1985).
4. M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 110-118 (10-13 October 1983). In *ACM Operating Systems Review* 17:5
5. E. R. Zayas, "Implementation and Evaluation of a Process Migration Facility," Thesis proposal, CMU, Computer Science Dept. (March 1984).
6. R. Finkel, M. Solomon, D. DeWitt, and L. Landweber, "The Charlotte Distributed Operating System: Part IV of the first report on the crystal project," Computer Sciences Technical Report #502, University of Wisconsin–Madison (October 1983).
7. M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," *Computer Network Performance Symposium*, pp. 47-55 (April 1982).
8. M. Livny, "The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems," Ph.D. thesis, Weizmann Institute of Science, Israel (August 1983).
9. R. M. Bryant and R. A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc. Second International Conference on Distributed Computing Systems*, pp. 314-323 (April 1981).
10. J. Stankovic, "Stability and Distributed Scheduling Algorithms," *IEEE Trans. on Software Engineering* **SE-11**(10) pp. 1141-1152 (October 1985).
11. P. Krueger and R. Finkel, "An Adaptive Load Balancing Algorithm for a Multicomputer," Computer Sciences Technical Report #539, University of Wisconsin–Madison (April 1984).
12. H-Y. Chang and M. Livny, "Priority in Distributed Systems," *Proc. IEEE 1985 Real-Time Symposium*, pp. 123-132 (December 1985).
13. R. A. Finkel, A. P. Anantharaman, S. Dasgupta, T. S. Goradia, P. Kaikini, C-P Ng, M. Subbarao, G. A. Venkatesh, S. Verma, and K. A. Vora, "Experience with Crystal, Charlotte, and Lynx," Computer Sciences Technical Report #630, University of Wisconsin–Madison (February 1986).
14. D. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," Technical Report 553, University of Wisconsin–Madison Computer Sciences (September 1984). To appear, IEEE Transactions on Software Engineering
15. Y. Artsy, H-Y Chang, and R. Finkel, "Charlotte: design and implementation of a distributed kernel," Computer Sciences Technical Report #554, University of Wisconsin–Madison (August 1984).
16. Y. Artsy, H-Y Chang, and R. Finkel, "Interprocess communication in Charlotte," *IEEE Software*, (July 1986). Accepted subject to revision
17. R. Cook, R. Finkel, D. DeWitt, L. Landweber, and T. Virgilio, "The Crystal nugget: Part I of the first report on the Crystal project," Technical Report 499, Computer Sciences Department, University of Wisconsin (April 1983).

18. M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," *Proc. 7th Symposium on Operating Systems Principles*, pp. 108-114 (December 1979).

19. F. Baskett, J. H. Howard, and J. T. Montague, "Task communication in Demos," *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pp. 23-31 (November 1977).

20. N. Wirth, "Modula: A language for modular multiprogramming," *Software — Practice and Experience* 7(1) pp. 3-35 (1977).

21. A. B. Barak and A. Shiloh, "A Distributed Load Balancing Policy for a Multicomputer," *Software — Practice and Experience* **15**(8) pp. 725-737 (August 1985).