Designing a process migration facility: The Charlotte experience

Yeshayahu Artsy	Raphael Finkel
Digital Equipment Corporation	Computer Science Department
550 King Street	University of Kentucky
Littleton, Massachusetts 01460	Lexington, KY 40506-0027

1998 addresses

raphael@cs.uky.edu y-artsy@msn.com

This paper was published in *IEEE Computer* **22:** 9, pp. 47-56, September 1989.

Key words: Operating Systems, Distributed Systems, Process Migration, System Design

Abstract

Our goal in this paper is to discuss our experience with process migration in the Charlotte distributed operating system. We also draw upon the experience of other operating systems in which migration has been implemented. A process migration facility in a distributed operating system dynamically relocates processes among the component machines. A successful process migration facility is not easy to design and implement. Foremost, a general-purpose migration mechanism should be able to support a range of policies to meet various goals, such as load distribution, and improved concurrency, and reduced communication. We discuss how Charlotte's migration mechanism detaches a running process from its source environment, transfers it, and attaches it into a new environment on the destination machine. Our mechanism succeeds in handling communication and machine failures that occur during the transfer. Migration does not affect the course of execution of the migrant nor that of any process communicating with it. The migration facility adds negligible overhead to the distributed operating system and provides fast process transfer.

1. Introduction

Our goal in this paper is to discuss our experience with process migration in the Charlotte distributed operating system. We identify major design issues and explain our implementation choices. We also contrast these choices with other migration implementations in the literature. This paper provides insights both into the specific task of implementing process migration for distributed operating systems and into the more general task of designing such systems.

A **preemptive process migration** facility dynamically relocates running processes between peer machines in a distributed system. Such relocation has many advantages. Studies have shown that it can be used to cope with dynamic fluctuations in loads and service needs[1], to meet real-time scheduling deadlines, to bring a process to a special device, or to improve the fault-tolerance of the system. Yet successful process migration facilities are not commonplace in distributed operating systems[234567]. The reason for this paucity is the inherent complexity of such a facility and the potential execution penalty if the migration policy and mechanism are not tuned correctly. It is not surprising that some operating systems prefer to terminate remote processes rather than rescue them by migration.

We can identify several reasons why migration is hard to design and implement. The mechanism for moving processes must be able to detach a migrant process from its source environment, transfer it with its context (the per-process data structures held in the kernel) and attach it in a new environment on the destination machine. These actions should complete reliably and efficiently. Migration may fail in case of machine and communication failures, but it should do so completely. That is, the effect should be as if the process was never migrated at all, or at worst as if the process had terminated due to machine failure. A wide range of migration policies might be needed, depending on whether the main concern is load sharing (avoiding idle time on one machine when another has a non-trivial work queue), load balancing (such as keeping the work queues of similar length), or application concurrency (mapping application processes to machines in order to achieve high parallelism). Policies may need elaborate and timely state information, since otherwise unnecessary process relocations may inflict performance degradation on both the migrant process and the entire system. The mechanisms to support different policies might differ significantly. If several policies are used under different circumstances, the migration mechanism must be flexible enough to allow policy modules to switch policies. The migration mechanism cannot be completely separated from process scheduling, memory management, and interprocess communication. Nevertheless, one would prefer to keep mechanisms for these activities as separate from each other as possible, to allow more freedom in testing and upgrading them. The fact that a process has moved should be invisible both to it and its peers, while at the same time interested users or processes should be able to advise the system about desired process distribution.

The process migration facility implemented for Charlotte is a fairly elaborate addition to the underlying Charlotte kernel and utility-process base. It separates policy (when to migrate which process to what destination) from mechanism (how to detach, transfer,

and reattach the migrant process). While the mechanism is fixed in the kernel and one of the utilities, the policy is relegated to a utility and can be replaced easily. The kernel provides elaborate state information to that utility. The mechanism allows concurrent multiple migrations and premature cancellation of migration. It leaves no residual dependency on the source machine for the migrant process. The mechanism copes with all conceivable crash and termination scenarios, rescuing the migrant process in most cases.

The next section presents an overview of Charlotte, and Section 3 presents its process migration facility. In Section 4 we discuss the issues encountered in building Charlotte's migration facility that have general application for any such facility. In discussing each issue, we present alternative design approaches adopted by other process migration facilities. We leave out the discussion of specific migration policies, as they are beyond the scope of this paper. We hope this account will give assistance to others contemplating adding a process migration facility, as well as advice to those designing other operating system facilities that may interact with later addition of process migration. We conclude with a brief list of concrete lessons and suggestions.

2. Charlotte overview

Charlotte is a message-based distributed operating system developed at the University of Wisconsin for a multicomputer composed of 20 VAX-11/750 computers connected by a token ring[9]. Each machine in Charlotte runs a kernel responsible for simple short-term process scheduling and a message-based inter-process communication (IPC) protocol. Processes are not swapped to backing store. A battery of privileged processes (**utilities**) runs at the user level to provide additional operating systems services and policies. The kernel and some utilities are multithreaded.

Processes communicate via **links**, which are capabilities for duplex communication channels. (The high-level language Lynx[10] actually hides this low-level mechanism and provides a remote procedure call (RPC) interface.) The processes at the two ends of a link may both send and receive messages by using non-blocking service calls. A process may post several such requests and await their completion later; it may cancel a pending request before that request completes. A link may be destroyed or given away to another process even during communication. In particular, a link is automatically destroyed when the process holding its other end terminates or its machine crashes; the process holding the local link end is so notified by the kernel. The protocol that implements communication semantics is efficient but quite complex[11]. It depends on full, up-to-date link information in the kernels of both ends of each link. Processes are completely unaware of the location of their communicating partners. Instead, they establish links to servers by having other processes (their parents or a name server utility) provide them.

Utility processes are distributed throughout the multicomputer, cooperating to allocate resources, provide file and connection services, and set policy. In particular, the **KernJob** (**KJ**) utility runs on each machine to provide a communication path between the local kernel and non-local processes. The **Starter** utility creates processes, allocates memory, and dictates medium-term scheduling policy. Each Starter process controls a subset of the machines; it communicates with their kernels (directly or via their KJs) to

receive state information and specify its decisions.

3. Process migration in Charlotte

Charlotte was designed as a platform for experimentation with distributed algorithms and load distribution strategies. We added the process migration facility in order to better support such experiments. Equally important, we wanted to explore the design issues that process migration raises in a message-based operating system. Figure 1 shows the effect of process migration. For convenience, throughout the paper we call the kernel on the source and destination machines **S** and **D**, respectively, and use **P** to represent the migrant process. During transfer, **P**'s process identifier changes, and the kernel data structures for it are completely removed from **S**, but the transfer is invisible to both **P** and its communication partners.

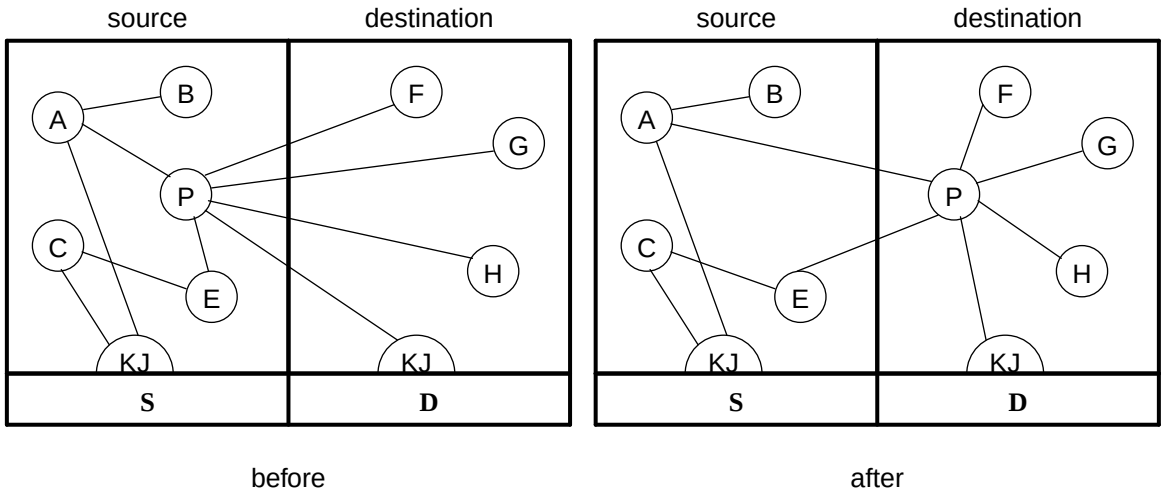

Figure 1: Example of migration

As shown, **P**'s links are relocated to the new machine. All processes continue to name their links the same after migration; they are unaware that link descriptors have moved sites and that local communication (performed in shared memory) has become remote communication (sent over the wire) and vice versa, and they see no change in message flow.

3.1. Policy

Migration policy is dictated by Starter utility processes. They base their decisions on statistical information provided by the kernels they control and on summary information they exchange among themselves. In addition, Starters accept advice from privileged utilities (to allow manual direction of migration and to enable or disable automatic control). When messages carrying statistics, advice, or notice of process creation or termination arrive, the Starter executes a policy procedure. (Introducing migration into the Starter only required writing that policy procedure and invoking it at the right times.) The policy procedure may choose to send messages to other Starters or to request some source kernel to undertake migration. Such requests are sent to the KernJob residing on the source machine to relay to its kernel. As discussed later, this approach adds

insignificantly to the cost of migration (a few procedure calls and perhaps a round-trip message), while it allows policies that integrate scheduling and memory allocation as well as local, clustered, or global policies.

3.2. Mechanism

The migration mechanism has two independent parts: collecting statistics and transferring processes. Both parts are implemented in the kernel.

Statistics include data on machine load (number of processes, links, CPU and network loads), individual processes (age, state, CPU utilization, communication rate), and selected active links (packets sent and received). These statistics are intended to be comprehensive enough to support most conceivable policies. We collect statistics in the following way.

Condition	**Action**
Significant event: message sent or received, data structure freed process created or terminated	Increment associated count
Interval passes	Sample process states and CPU, network loads
Period of *n* intervals passes	Summarize data, Send to starter

To balance accuracy with overhead, we used in our tests an interval of 50 to 80 ms and a period of 100 intervals (5 to 8 seconds). The overhead for collecting statistics was less than 1% of total cpu time.

Transferring processes occurs in three phases.

(1) **Negotiation**. After being told by their controlling Starter processes to migrate **P**, **S** and **D** agree to the transfer and reserve required resources. If agreement cannot be reached, for example because resources are not available, migration is aborted and the Starter that requested it is notified.

(2) **Transfer**. **P**'s address space is moved from the source to the destination machine. Meanwhile, separate messages are sent to each kernel controlling a process with a link to **P** informing that kernel of the link's new address.

(3) **Establishment**. Kernel data structures pertaining to the migrant process are marshaled, transferred, and demarshaled. (Marshaling requires copying the structure to a byte-stream buffer, and converting some data types, particularly pointer types.) No information related to the migrant is retained at the source machine.

Process-kernel interface

We added four kernel calls to the process-kernel interface.

`Statistics(What : action; Where : address)`
: The KernJob invokes this call (on behalf of a Starter) so that the kernel will start collecting statistics and placing them in the given address (in the KernJob virtual space). The call can also be used to stop statistics collection.

`MigrateOut(Which : process; WhereTo : machine)`
: This call enables the Starter (or its KernJob proxy, if the Starter resides on another machine) to initiate a migration episode.

`Boolean; Memory : list of physical regions)`
`MigrateIn(Which : process; WhereFrom : machine; Accept :`
: The Starter (or its KernJob proxy) uses this call to approve or refuse a migration from the given machine to the machine on which the call is performed. If Starters have negotiated among themselves, the Starter controlling the destination machine may approve a migration even before the one controlling the source machine calls `MigrateOut`. The Memory parameter tells the kernel where in physical store to place the segments that constitute the new process. (The Starter learns the segment sizes either through negotiation with its peer or from **D**'s request to approve a migration offer received from **S**.)

`CancelMigration(Which : process; Where : machine)`
: The Starter invokes this call to abort an active `MigrateIn` or `MigrateOut` request. This call is rejected if the migration has already reached a commit point.

None of these calls blocks the caller. The kernel reports the eventual success or failure of the request by a message back to the caller.

Mechanism details

Three new modules were created in the kernel to implement the migration mechanism. The migration interface module deals with the new service calls from processes. The migration protocol module performs the three phases listed above. The statistics module collects and reports statistics. These modules are invoked by two new kernel threads. The statistician thread awakens at each interval to sample, or average and report statistics to the Starter. A process-receiver thread starts in **D** for each incoming migrant process. It uses a simpler and faster communication protocol than that used by ordinary IPC.* However, negotiation and other control messages use the ordinary communication protocol and are funneled through the IPC queues in order to synchronize process and link activities.

Figure 2 shows both high- and low- level negotiation messages. In our example, the left Starter process controls machine 1, and its peer controls machine 3. The first two messages represent a Starter-to-Starter negotiation that results in deciding to migrate process **P** from machine 1 to 3. Their decision is communicated to **S** in message 3, which is

*The standard protocol must expect extremely complex scenarios that cannot arise in this conversation and must employ link data structures that are not germane here. The cost of introducing a streamlined protocol was slight in comparison to the speed it achieved.

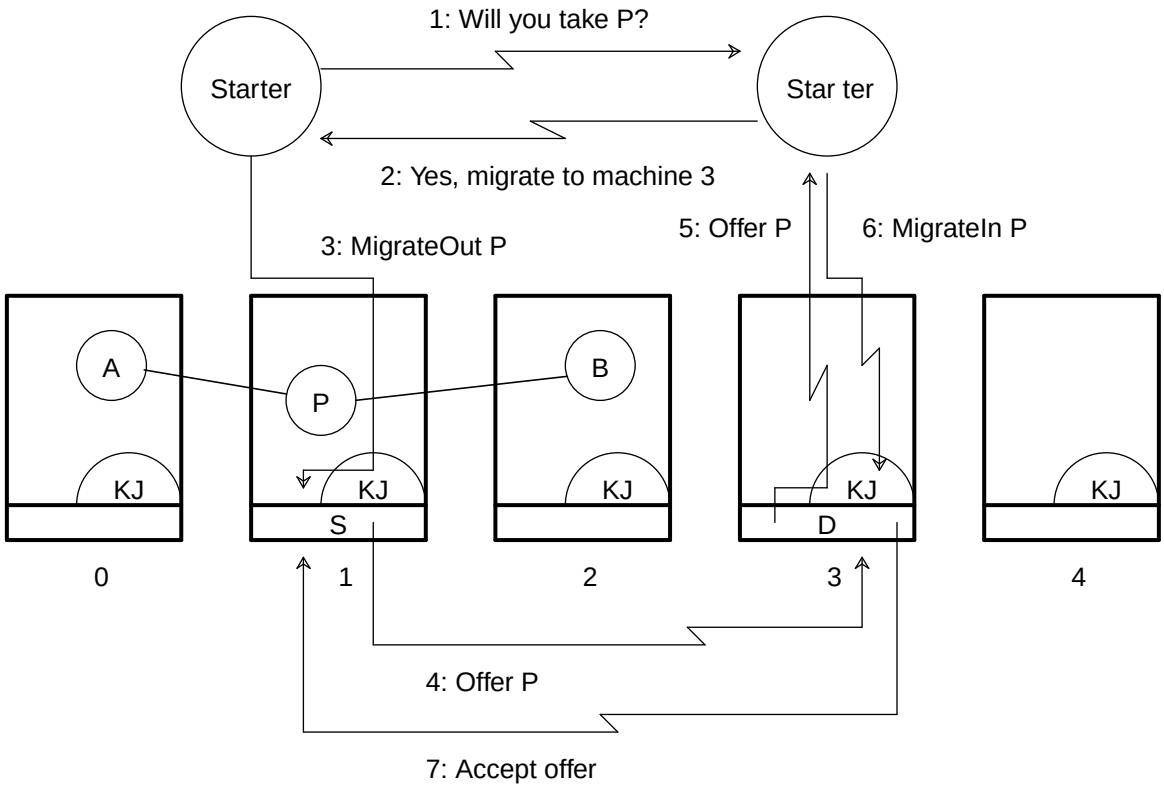

Figure 2: Negotiation phase

either a direct service call (if the Starter runs on machine 1) or a message to the KernJob on machine 1 to be translated into a service call. **S** then offers to send the process to **D**. The offer includes **P**'s memory requirements, its age, its recent CPU and network use, and information about its links. If **D** is short of the resources needed for **P**, or if too many migrations are in progress, it may reject the offer outright. Otherwise, **D** relays the offer to its controlling Starter (message 5). The relay includes the same information as the offer from **S**. We relay the offer to let the policy module reject a migrant at this point. Although that Starter may have already agreed to accept **P** (in message 2), it may now need to reject the offer due to an increase in actual or anticipated load or lack of memory. Furthermore, the Starter must be asked because the kernel has no way to know if it has even been consulted by its peer Starter, and the Starter must allocate memory for the migrant. The Starter's decision is communicated to **D** by a `MigrateIn` call (message 6). No relay occurs if the Starter has already called `MigrateIn` to preapprove the migration. Before responding to **S** (message 7), **D** reserves necessary resources to avoid deadlock and flow-control problems. Preallocation is conservative; it guarantees successful completion of multiple migrations at the expense of reducing the number of concurrent incoming migrations.

After message 7 is sent, **D** has committed itself to the migration. If **P** fails to arrive and the migration has not been cancelled by **S** (see next), then the machine of **S** must be down or unreachable. **D** discovers this condition through the standard mechanism by

which kernels exchange “heart-beat” messages and reclaims resources and cleans up its state.

When message 7 is received, **S** is also committed and starts the transfer. Before each kernel commits itself, its Starter can successfully cancel the migration, in which case **D** replies *Rejected* to **S** (in message 7), or **S** sends *Regretted* to **D** (not shown). The latter also occurs if **P** dies abruptly during negotiation. To separate policy from mechanism, **S** does not retry a rejected migration unless so ordered by its Starter.

Figure 3 shows the transfer phase. **S** concurrently sends **P**’s virtual space to **D** (message 8) and link update messages (9) to the kernels controlling all of **P**’s peers. Message 8 is broken into packets as required by the network. **D** has already reserved physical store for them, so the packets are copied directly into the correct place. Message 9 indicates the new address of the link; it is acknowledged (not shown) for synchronization purposes. After this point, messages sent to **P** will be directed to the new address and buffered there until **P** is reattached. Kernels that have not received message 9 yet may still continue to send messages for **P** to **S**. Failure of either the source or the destination machine during this interval leaves the state of **P** very unclear. Since it would require a very complex protocol (sensitive to further machine failures) to recover **P**’s state, we opted to terminate **P** if one of these machines crashes at this stage.

Finally, **S** collects all of **P**’s context into a single message and sends it to **D** (message 10). This message includes control information, the state of all of **P**’s links, and details of communication requests that have arrived for **P** since transfer began. Pointers in **S**’s data structures are tracked down, and all relevant data are marshaled together. **D** demarshals

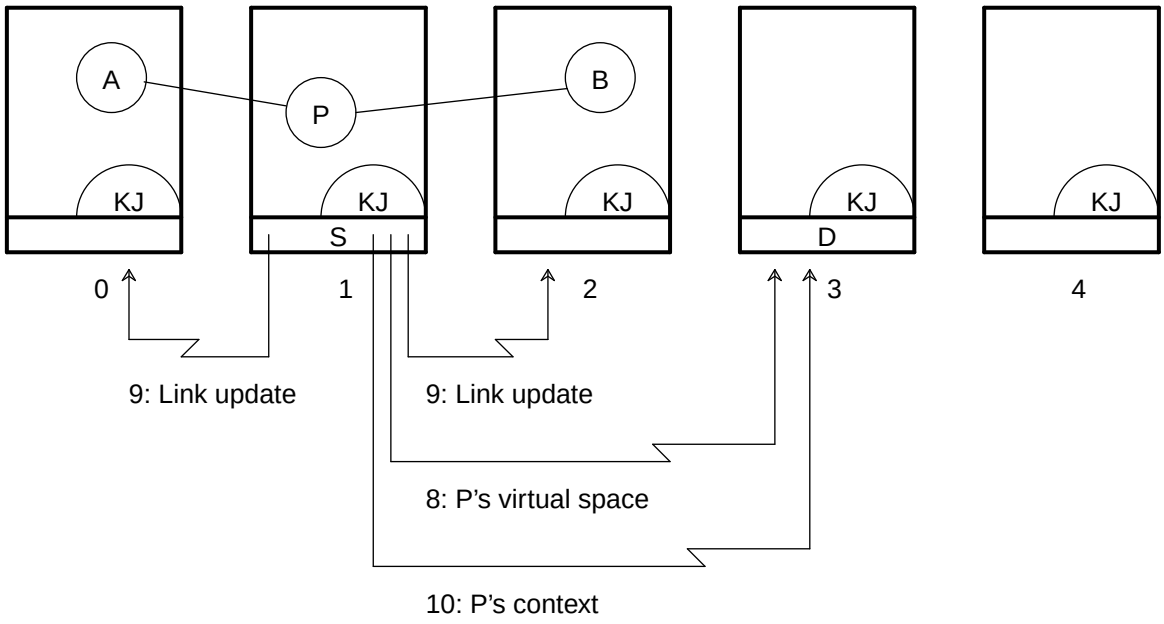

Figure 3: Transfer phase

the message into its own data structures.

Although it is conceptually simple, the transfer stage is actually quite complex and time-consuming, mainly because Charlotte IPC is rich in context. Luckily, our design saves the migration mechanism from dealing with messages in transit to or from **P**. Since the kernel provides message caching but no buffering, a message remains in its sender's virtual address space until the receiver is ready to accept it or until the send is cancelled. Hence, **S** does not need to be concerned with **P**'s outgoing messages; likewise, it may drop from its cache any message received for **P** that **P** has not yet received. Such a message will be requested by **D** from the sender's kernel when **P** requests to receive it. The link structures sent in message 9 clearly indicate which links have pending sent or received messages. Another advantage in our design is that we do not have to alter or transfer **P**'s context maintained by distributed utilites, such as open files, which are accessed via location-independent links.

The establishment phase is interleaved with transfer. Data structures are deallocated as part of marshaling, and the reserved ones are filled during demarshaling. After transfer has finished, **D** adjusts **P**'s links and pending events and inserts **P** into the appropriate scheduling queue. Those communication requests that were postponed while **P** was moving have been buffered by **S** and **D**; they are now directed to the IPC kernel thread in **D** in their order of arrival. (For each link, all those buffered at **S** precede those buffered at **D**.) Their effect on **P** is not influenced by the fact that it has moved. Finally, the Starter and KernJob processes for both the source and destination machine are informed that migration has completed so they can update their data structures appropriately. A failure of either of the two machines at the transfer phase is detected by the remaining one, which will abort the migration, terminate the migrant, and clean up its state.

3.3. Performance

We measured performance of migration in Charlotte on our VAX/11-750 machines connected by a Pronet token ring. The underlying mechanisms have the following costs. It takes 11 ms to send a 2 KB packet to another machine reliably via the general-purpose inter-machine communication package that Charlotte uses, 0.4 ms to switch context between kernel and process, 10 ms to transfer a single packet between processes residing on the same machine, and 23 ms to transfer a packet between processes residing on different machines.

We measured the average elapsed time to migrate a small (32 KB), linkless process as 242 ms (standard deviation σ = 2 ms), provided the Starter controlling **D** has preapproved the migration. Each additional 2 KB of image adds 12.2 ms to the migration time. The following formula fits our measurements of the average elapsed time spent in migration.

$$\text{Charlotte time} = 45 + 78p + 12.2s + 9.9r + 1.7q$$

p = 0 if **D**'s Starter has approved migration in advance;
else 1 if **D**'s Starter is not on the destination machine;
else about 0.2.
s = size of the virtual space in 2-KB blocks
r = 0 if all links are local; 1 otherwise
q = number of non-local links (1 if none).

These measures deviate by about 5% with different locations of the Starter and the overall load. This formula shows that it takes about 750ms to migrate a typical process of 100KB and 6 links (or 670ms if **D**'s Starter is local), and about 6 seconds for a large process of 1MB. Actual CPU time spent on the migration effort for a 32 KB process with no links is about 60 ms for **S** and about 32 ms for **D**. Table 1 shows how this time is spent.

S		**D**
5.0	Handle an offer	5.4 Handle an offer
2.6	Prepare 2 KB image to transfer	1.2 Install 2 KB of image
1.8	Marshal context	1.2 Demarshal context
6.9	Other (mostly kernel context switching)	4.7 Other

Table 1: Kernel time spent migrating a linkless 32 KB process

Each link costs **S** an additional 1.6 to 2.8 ms of CPU time to prepare link-update messages and to marshal relevant data structures. Collecting statistics requires about 1% of overall elapsed time, and another 2% of all time is spent delivering the statistics to the Starter. A production version of Charlotte, optimized and stripped of debugging code, could exhibit a significant speed improvement.

It is hard to compare Charlotte's migration performance with results published for other implementations, because each uses a different underlying computer, and each operating system dictates its own process structure. Nonetheless, to give the reader some form of comparison, we present formulas for migration speed under Sprite (Sun-3 workstations, about 4 times faster than our VAX-11/750 machines), V (Sun-2, about 2 times faster than our machines), and Accent (Perq workstations). These formulas are extrapolations from a few measurement points reported elsewhere[467].

$$\text{Sprite time} = 200 + 3.6s + 14f$$

$$\text{V time} = 80 + 6s$$

$$\text{Accent time} = 1180 + 115s$$

s = size of the virtual space in KB
f = number of open files

In particular, a "typical" 100KB process would be transferred in about 560ms in Sprite, 680ms in V, and perhaps 12.7 seconds in Accent. To migrate a large, 1MB process would take at least 3.8 seconds in Sprite, 6 seconds in V, and 116 seconds in Accent. In Accent, sending the context of a process occupies about 1 full second. The virtual space is sent later on demand, so the full cost of transfer is spread over a long period, but part of this

cost is saved if not all the pages are referenced. V precopies the address space while the process is still running, so the lost time suffered by the process is quite short.

4. Design issues

Designing a process migration facility requires that one consider many complex issues. We will discuss the separation of policy and mechanism, the interplay between migration and other mechanisms, reliability, concurrency, the nature of context transfer, and to what extent processes should be independent of their location. These issues interrelate to one another, so the following discussion will occasionally need to postpone details until later sections. Moreover, the approaches that we and others adopt to various problems depend somewhat on the design of other components of the operating system. Due to space limitations, we do not discuss these dependencies in detail.

4.1. Structure

The first step in designing a process migration facility is to decide where the policy-making and mechanism modules should reside. We believe that this decision is of major importance since it cannot be easily reversed, unlike most of the design of the migration protocol. Communication-kernel operating systems tend to put mechanism in the kernel and policy in trusted utility processes. In the case of process migration, mechanism is intertwined with both short-term scheduling and IPC, so it fits best in the kernel. Policy, on the other hand, is associated with long-term scheduling and resource management, so it fits well in a utility process. Several considerations affect the success of separation: how efficient the result is, how adequately it provides the needed function, and how conceptually simple are the interfaces and the implementation.

Efficiency and Simplicity

The principal reason one might place policy in the kernel instead of in a utility is to simplify and speed up the interface between policy and mechanism. Any reasonable policy depends on statistics that are maintained primarily in the kernel. High quality decisions may well require large amounts of accurate and comprehensive data. Placing policy outside the kernel incurs execution overhead and latency in passing these statistics in one direction and decisions in the other.

Our experience with Charlotte, however, shows that placing the policy in a utility results in a net efficiency *gain*. Although separation incurs the extra cost of one message for statistics reporting and one kernel call (and perhaps another message round-trip) for decision reporting, it allows reduction of communication and more global policy due to the fact that each Starter process decides policy for a set of machines. As to the latency in passing statistics and decisions, studies have found that good policies tend to depend mostly on aggregate and medium-term conditions, ignoring short-term conditions or small delays.

The designer may choose to support only simple policies, in which case they may well be put in the kernel. For example, the migration policy in V[6] and Sprite[4] is mostly manual, choosing a remote idle workstation for a process or evicting it when the station's

owner so requires. In systems where migration is used to meet real-time scheduling deadlines, policy tends to be simple or very sensitive to even small delays, and hence could or should be placed in the kernel. Integrating the policy in the kernel, however, might obstruct later expansion or generalization.

We can achieve conceptual separation of policy and mechanism without incurring a large interface cost by assigning them to separate layers that share memory. MOS[2] adopts this approach by dividing the kernel to two layers, one to implement migration mechanism and other low-level functions, and the other layer to provide policy. These layers share data structures and communicate by procedure calls. Although such sharing improves efficiency, it becomes harder to modify policy, since changes require kernel recompilation, and inadvertent errors are more serious.

Function and flexibility

Placing policy outside the kernel facilitates testing diverse policies and choosing among policies tuned for different goals, such as load sharing, load balancing, improving responsiveness, communication-load reduction, and placing processes close to special devices. Being able to modify policy is especially important in an experimental environment. Our students needed only a few hours to learn the interface and major components of the Starter in order to start trying different policies; they did not need to learn peculiarities of the kernel or of the migration mechanism. This flexibility would be impossible if policy were embedded in the kernel.

In various distributed systems, such as Demos/MP[5], Accent[12], and Charlotte, resource-management policies are often relegated to utilities. Putting migration policy in those same processes can allow more integration and coordination of the policies governing the system.

The designer of process migration should be aware of the danger of separating policy and mechanism too far. Letting policy escape from trusted utility servers into application programs may result in performance degradation or even thrashing. This problem occurs, for example, if applications may decide the initial placement and later relocation of their processes, as in Locus, without getting any assistance from the kernel in the form of timely state and load information.

4.2. Interplay between migration and other mechanisms

The process migration mechanism can be designed independently of other mechanisms, such as IPC and memory management. The actual implementation is likely to see interactions among these mechanisms. However, design separation means that the migration protocol should not change when the IPC protocol does. In Charlotte, for example, we did not change the IPC to add process migration, nor did migration change when we later modified the semantics of two IPC primitives. In contrast, we had to change the marshaling routines when an IPC data structure changed.

We feel that ease of implementation is a dominant motivation for separating mechanisms from each other when process migration is added to an existing operating system, such as was the case in Demos/MP, Charlotte, V, and Accent. A secondary motivation is

that the migration code can be deactivated without interfering with other parts of the kernel. In Charlotte, for instance, we can easily remove all the code and structures of process migration at compile time or dynamically turn the mechanism on and off. In contrast, efficiency arguments would favor integrating all mechanisms. Accent, for example, uses a transfer-on-reference approach to transmitting the virtual space of the migrant process that is based on its copy-on-write memory management. If process migration is intended from the start, as in MOS and Sprite, integration can reduce redundancy of mechanisms. In retrospect, Charlotte would have used a different implementation for IPC if the two mechanisms had been integrated from the start. We would have used **hints** for link addresses, which are inaccurate but can be readily checked and inexpensively maintained, rather than using **absolutes**, whose complete accuracy is achieved at a high maintenance cost.

Some interactions seem to be necessary. In Charlotte, for instance, we chose to simplify the migration protocol by refusing to migrate a process engaged in multi-packet message transfer. We therefore depend slightly on knowledge of the IPC mechanism to avoid complex protocols. Similarly, both MOS and Sprite refuse to migrate a process engaged in RPC until it reaches a convenient point, which may not happen for a long time. Other interactions make sense in order for process migration to take advantage of existing facilities. For example, Locus uses existing process-creation code to assist in process migration.

4.3. Reliability

Migration failures can occur due to network or machine failure. The migration mechanism can simply ignore these possibilities (as does Demos/MP) in order to streamline protocols. The Charlotte implementation is able to rescue the migrant from many failures by several means. First, it transfers responsibility for the migrant as late as possible, to survive failure of the destination or the network. Second, it detaches the migrant completely from its source, to survive later failures there. Third, the migrant is protected from failures of other machines; at most, some of its links are automatically destroyed if the machine where their other ends reside has crashed. Rescuing migrating processes under all failure circumstances requires complex recovery protocols, and most likely large overhead for maintaining process replicas, checkpoints, or communication logs. We were unwilling to pay that cost in Charlotte. Instead, we terminate the migrant if either the source or destination machine crashes during the sensitive time of transfer when messages for the migrant may have arrived at either machine, as discussed earlier. Modifying our IPC to use hints for link addresses, as mentioned above, would have made this step less fragile.

4.4. Concurrency

Various levels of concurrency are conceivable:

- Only one migration in the network at a time
- Only one migration affecting a given machine at a time

- No constraints on the number of simultaneous migrations

The Charlotte mechanism puts no constraint on concurrency. Restricting process migration can make the mechanism simpler, especially in operating systems using a connection-based IPC. The most restrictive alternative guarantees that the peers of the migrant process are stationary, so redirection of messages is straightforward. It also tends to mitigate policy problems of migration thrashing, flooding a lightly-loaded machine with immigrants, and completely emptying a loaded machine.

Enforcing such a constraint, on the other hand, requires arbitrating contention, which can be expensive. In addition, limiting concurrency constrains policies that otherwise would be able to evacuate a failing machine quickly or react immediately to a severe load imbalance. We therefore believe that the policy problems alluded to above should be solved by policy algorithms, not by a limitation imposed by the mechanism.

Allowing simultaneous migrations introduces the peculiar problem of name and address consistency: ensuring that all processes and kernels have a consistent view of the world. The problem is manifest in operating systems like Charlotte, in which communication is carried out over established channels and kernels require up-to-date location information. If two processes connected by a channel migrate at the same time, their kernels may have false conception of the remote channel ends. The problem is not critical in operating systems that treat communication addresses as hints, such as V, because communication encountering a hint fault will restore the hint by invoking a process-finding algorithm. This solution incurs execution and latency costs as messages are transmitted. Where absolutes are used, forwarding pointers, such as those used in Demos/MP, may solve the problem, but they introduce long-lived residual dependencies. In Charlotte, we send link-address updates before migration completes, and we buffer notifications for messages arriving during the transfer. The immediate acknowledgement of the updates, even when the other link end is simultaneously given away or migrating, prevents deadlock. When migration completes, **D** processes the notifications buffered by the two kernels and regains a consistent view of **P**'s links, even if their remote ends have moved meanwhile.

Within a single source or destination, we could restrict concurrency to one migration attempt at a time. This restriction simplifies the kernel state and again reduces risks of thrashing. However, complexity can be reduced by creating a new kernel thread for each migration in progress, executing a finite-state protocol independently of other migration efforts. Using these techniques, we found that allowing concurrent migrations in the same machine incurs only a small space overhead and minor execution costs.

4.5. Context transfer and residual dependency

At some point during migration, the process must be frozen to ensure a consistent transfer.

What and when to freeze

Three activities need to be frozen: (1) process execution, (2) outgoing communication, and (3) incoming communication. The first two activities are trivial to freeze.

Freezing incoming communication can be accomplished by (a) telling all peers to stop sending, (b) delaying incoming messages, or (c) rejecting incoming messages. Option (a) requires a complex protocol if concurrent migrations are supported or if crashes must be tolerated. Option (c) requires that the IPC be able to resend rejected messages, as in V. In Charlotte, we chose option (b) because it seems the simplest and because it does not interfere with other mechanisms.

Very early freezing (for example, when a process is considered as a migration candidate) has the advantage that the process does not change state between the decision and migration. Otherwise, the migration decision may be worthless, since the process could terminate or start using resources differently. However, freezing a process hurts its response time, which flies in the face of one of the goals of migration. Less conservatively, we can freeze a process when it is selected as a candidate, but before the destination machine has accepted the offer. Even less conservative alternatives include freezing at the point migration is agreed upon, or even when it is completed. Each more liberal choice increases the process' responsiveness at the cost of protocol complexity.

In Charlotte, we chose to balance responsiveness and protocol simplicity by freezing both execution and communication only when context is marshaled and transferred. We delay incoming communication by buffering input notifications at **S** and both notifications and data at **D** until **P** is established. We verified (by exhaustive enumeration of states in our automata that drive the IPC protocol) that the ensuing delays could not cause deadlock or flow control problems[11]. In this way, a minimal context is transferred during negotiation (such as how many links **P** has and where their ends are); the final transfer reflects any change in **P**'s state during migration.

MOS and Locus freeze the migrant earlier, when it is selected for migration. V, in contrast, freezes a process for a minuscule interval near the end of transfer. While transfer is in progress, the migrant continues to execute; pages dirtied during that episode are sent again in another transfer pass, and so forth until a final pass. Incoming messages are rejected during the short freeze, with the understanding that the IPC mechanism will timeout and retransmit them. The result is that the migrant suffers a delay comparable to that required to load a process into memory[6].

Redirecting communication

Redirecting communication requires that state information relevant to the communication channels be updated and that peer kernels discover the migrant's new location. In a connectionless IPC mechanism, a process holds the names of its communication peers. For example, V processes use process identifiers as destinations[13]. To redirect communications in such an environment, a kernel may broadcast the new location. Broadcast can be expensive for large networks with frequent migrations. Alternatively, peers can be left with incorrect data that can be resolved on hint faults. Another alternative is to assign a home machine to each process; the home machine always knows where the process is. Locus uses this method to find the target of a signal. Sprite is similar; the home machine manages signals and other location-dependent operations on behalf of the migrant. Of course, resorting to a home machine makes communication failures more likely and

sharply increases the cost of certain kernel calls.

In a connection-based IPC environment with simplex connections, such as Accent and Demos/MP, the kernel of the receiving end of a connection does not know where the senders are. That means that **S** cannot tell which kernels to inform about **P**'s migration. Instead, a forwarding pointer may be left on **S** to redirect new messages as they arrive. Demos/MP uses this strategy. Another approach is to introduce a stationary "middleman" between two or more mobile ends of a connection. In Locus, cross-machine pipes may have several readers and writers, but they have only one fixed storage site. When a reader or writer migrates, the kernel managing the storage site is informed. In Charlotte, the duplex nature of links suggests maintaining information at both ends about each other, so **S** can tell all peers that **P** has moved. Transferring these link data along with **P**, though, incurs marshaling, transmission, and demarshaling overhead.

Residual dependency

The migrant process can start working on the destination machine faster if it can leave some of its state temporarily on the source machine. When it needs to refer to that state, it can access it with some penalty. To reduce the penalty, state can be gradually transferred during idle moments. State can also be pulled upon demand. The choice between moving the entire address space or only a part is reminiscent of the controversy in network file systems whether entire files should be transferred or only pages for remote file access. Locality of execution suggests transferring at least the working set of **P** during migration, and the rest when needed. On the other hand, the objective of residual independence suggests removing any trace of **P** from the source machine.

In MOS, virtually the entire state of **P** could remain in the source machine, since **D** can make remote calls on **S** for anything it needs. For efficiency reasons, however, MOS transfers most of **P**'s context when it migrates. In Sprite, part of **P**'s context always resides in its home machine, but none is left on the source machine when it is evicted. This approach costs about 15 ms to demand-load a page and perhaps 4 ms to execute some of the kernel calls remotely (about 9-fold increase). In Accent, processes do not make kernel calls directly, but rather send messages to a kernel port. Therefore, no state needs to be moved with a process; it can all remain with **S** and be accessed as needed by kernel calls to the old port. In addition, Accent implements a lazy transfer of data pages on demand. Similarly, in Sprite, **S** acts as a paging device for **D**. These approaches trade efficiency of address-space transfer for risks of machine unavailability, protocol complexity, and later access penalties.

4.6. Location independence

Many distributed operating systems adhere to the principle of location transparency. In particular, process names are independent of their location, processes can request identical kernel services wherever they reside, and they can communicate with their peers equally well (except for speed) wherever they might be. The principle of location transparency must be followed carefully to enable migration. Migration requires that naming schemes be uniform for local and remote communication and that resource references not

depend on the host machine. For example, Charlotte objects are all named by the links that connect a client to them. When a process moves, the names it uses for its links are unchanged, even though **D** remaps them to different internal names. The fact that local communication is treated differently from remote communication is localized in a few places in the kernel. Processes may have pointers or indices to kernel data structures, but those are maintained by the kernel. The actual data structures, pointers and indices are remapped invisibly during migration. If such values were buried inside the processes' address spaces, migration would be impossible or extremely complicated. Sprite maintains location transparency throughout multiple migrations by keeping location-dependent information on **P**'s home machine and by directing some of **P**'s kernel calls there.

Transaction management and multithreading also pose transparency problems. A transaction manager must not depend on the location of its clients. Multithreaded processes must be moved *in toto*. If threads may cross address spaces, the identity of one thread may be recorded in several address spaces, leading to location dependencies.

Of course, any policy setter, such as the Charlotte Starter, needs to know the location of all processes and perhaps the endpoints of their heavily-used links. Making this information available need not compromise the principle of transparency. The policy module does not use this information to send messages, only to inform itself about decisions it needs to make. Likewise, for the sake of openness, a design may allow processes willing to participate in migration decisions to receive location information and contribute migration advice.

5. Conclusions

Our experience with Charlotte and others' experience with Sprite, V, MOS, and Demos/MP, show that process migration is possible, if not always pleasant. We found that separating the modules that implement mechanism from those responsible for policy allows more efficient and flexible policies and simplifies the design. Migration interact with other parts of the kernel. In particular, the implementation shares structures and low-level functions with other mechanisms. Nonetheless, we found it possible to keep the mechanisms fairly independent of each other, gaining high code modularity and ease of maintenance.

Software and hardware failures are a fact of life. Our migration protocol can rescue the migrant in most failure situations and restore the state in all of them, despite the fact that the migrant continues its interaction with other processes at early stages of migration. In some cases, though, we opt to kill the migrant even if rescue is dimly conceivable. We chose to postpone committing migration until late during the transfer itself (to deal with early destination crash), while removing any dependency of the migrant on the source as soon as migration completes (to deal with late source crash).

Except for potential confusion suffered by policy modules, it is not particularly hard to achieve simultaneous migrations, even those involving a single machine. The Charlotte IPC requires absolute state information, so we could not try to reduce the cost of migration by sacrificing accuracy. IPC mechanisms that use hints or are connectionless can shorten the elapsed time for migration but then probably pay more during

communication. Designs that require previous hosts to retain forwarding information for an arbitrary period after migration are overly susceptible to machine failure. Forwarding data structures, although small, tend to build up over time.

6. Acknowledgements

The design of process migration in Charlotte was inspired by discussions with Amnon Barak of the Hebrew University of Jerusalem in 1984. The authors are indebted to Cui-Qing Yang for modifying Charlotte utilities to support process migration and to Hung-Yang Chang for many fruitful discussions about the design. Andrew Black and Marvin Theimer provided helpful comments on an early draft, and the referees suggested many stylistic improvements. The Charlotte project was supported by NSF grant MCS-8105904 and DARPA contracts N00014-82-C-2087 and N00014-85-K-0788.

References

1. P. Krueger and M. Livny, "When is the best load sharing algorithm a load balancing algorithm?," Computer Sciences Technical Report #694, University of Wisconsin–Madison (April 1987).
2. A. B. Barak and A. Litman, "MOS: A Multicomputer Distributed Operating System," *Software — Practice and Experience* **15**(8) pp. 725-737 (August 1985).
3. D. A. Butterfield and G. J. Popek, "Network tasking in the Locus distributed UNIX system," *Proc. of the Summer USENIX conference*, pp. 62-71 USENIX Association, (June 1984).
4. F. Douglis and J. Ousterhout, "Process migration in the Sprite Operating System," *Proc. of the 7th Int'l Conf. on Distributed Computing Systems*, pp. 18-25 IEEE Computer Press, (September 1987).
5. M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," *Proc. of the Ninth ACM Symp. on Operating Systems Principles*, pp. 110-118 ACM SIGOPS, (October 1983). In *Operating Systems Review* 17:5
6. M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proc. of the Tenth Symp. on Operating Systems Principles*, pp. 2-12 ACM SIGOPS, (December 1985).
7. E. R. Zayas, "Attacking the process migration bottleneck," *Proc. of the Eleventh ACM Symp. on Operating Systems Principles*, pp. 13-24 ACM SIGOPS, (November 1987). In *Operating Systems Review* 21:5
8. D. A. Nichols, "Using idle workstations in a shared computing environment," *Proc. of the Eleventh ACM Symp. on Operating Systems Principles*, pp. 5-12 ACM SIGOPS, (November 1987). In *Operating Systems Review* 21:5
9. Y. Artsy, H-Y. Chang, and R. Finkel, "Interprocess communication in Charlotte," *IEEE Software* **4**(1) pp. 22-28 IEEE Computer Society, (January 1987).

10. M. L. Scott, “Language support for loosely coupled distributed programs,” *IEEE Trans. on Software Eng.* **SE-13**(1) pp. 88-103 IEEE, (January 1987).
11. 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).
12. R. F. Rashid and G. G. Robertson, “Accent: A communication oriented network operating system kernel,” *Proc. of the Eighth ACM Symp. on Operating Systems Principles*, pp. 64-75 ACM SIGOPS, (December 1981).
13. D. Cheriton, “The V Kernel: A software base for distributed systems,” *IEEE Software* **1**(2) pp. 19-42 (April 1984).