Process Migration for Heterogeneous Distributed Systems

Matt Bishop*
Department of Computer Science
University of California at Davis

Mark Valence*
Sassafras Software Inc.
Hanover, NH

Leonard F. Wisniewski†
Department of Computer Science
Dartmouth College

Dartmouth PCS-TR95-264

August 21, 1995

Abstract

The policies and mechanisms for migrating processes in a distributed system become more complicated in a heterogeneous environment, where the hosts may differ in their architecture and operating systems. These distributed systems include a large quantity and great diversity of resources which may not be fully utilized without the means to migrate processes to the idle resources. In this paper, we present a graph model for single process migration which can be used for load balancing as well as other non-traditional scenarios such as migration during the graceful degradation of a host. The graph model provides the basis for a layered approach to implementing the mechanisms for process migration in a Heterogeneous Migration Facility (HMF). HMF provides the user with a library to automatically migrate processes and checkpoint data.

1 Introduction

Distributed systems provide users with access to remote resources spread across a room, a community, or even a country. A *heterogeneous distributed system* connects host machines with different architectures and configurations. A wide area network (WAN) could include an extensive range of resources, e.g., a Cray T3D supercomputer, Silicon Graphics Indy workstations, and Macintosh personal computers. All host machines in a distributed system possess raw computing power, at the very least.

Load balancing[1] attempts to better utilize this raw computing power [WM85]. A static load balancing algorithm assigns a process to a host upon invocation and a dynamic load balancing algorithm assigns a process to another host after executing on its current host for some time.

*Supported in part by NASA under grant NAG2-628.

†Supported in part by a Dartmouth Fellowship, by NASA under grant NAG2-628, and by the National Science Foundation under Grant CCR-9308667. Author's e-mail address: `wisnie@cs.dartmouth.edu`.

[1]In this context, what we call load balancing shall include load sharing.

When load balancing dynamically, a distributed system must be able to *migrate* a process from its current host to a destination host. The *migration policy* determines how to balance the processing load across the hosts in the distributed system. That is, the migration policy decides when a migration should occur. The *migration mechanism* extracts a process and its associated context from its source host and establishes the process on its destination host for execution. The system designer can specify a migration policy and mechanism by answering the questions HOW, WHEN, and WHERE does a process migrate and WHO makes the decision to migrate.

Migration on a heterogeneous distributed system can be a difficult task with rich rewards. The existing distributed systems with the ability to migrate a process are homogenous, that is, all the processors in the distributed system are identical. The uniform architectures and operating systems of the hosts in the distributed systems simplify the design and implementation of migration policies and mechanisms. Since heterogeneous distributed systems are typically spread out over large geographic areas (e.g., wide area networks), they usually possess a large quantity of raw computing power as well as a greater diversity of resources [NBL$^+$88]. Migrating a process between two different host architectures and operating systems involves many complex tasks which include the translation and transfer of state as well as the coordination of the *source* and *destination* hosts.

We endeavour to examine the feasibility of providing support for migration in a heterogeneous distributed system. This paper provides the framework for the design and implementation of migration policies and mechanisms in the following ways.

1. We define a graph model of single process migration and model the various costs associated with the migration policy and mechanism. Unlike previous models, the system designer can use this model for scenarios other than load balancing to take advantage of the ability to migrate, such as graceful degradation, process evacuation, and real-time response.

2. We present the levels of abstraction by which to implement facilities for the migration of processes. The levels of abstraction differ in the amount of support that they provide for checkpointing and migrating the code and data of the executing program.

3. We provide the design and specification of the Heterogeneous Migration Facility (HMF), a library of routines to accommodate the migration of a process on a heterogeneous distributed system. HMF includes a preprocessor to convert datatypes into an architecture-independent representation.

The outline of this paper is as follows. Section 2 discusses the various mechanisms and some of the policies used by existing systems that support migration. In Section 3, we present the graph model for single process migration and model the costs of migration by answering the HOW, WHEN, WHERE, and WHO questions. Section 4 abstracts the levels to which migration may occur on the code and data of a program. This abstraction suggests a layered approach in the design of a migration facility such as HMF described in Section 5. Section 6 contains some concluding remarks.

2 Previous work

This section discusses significant features of existing facilities for migration on homogeneous distributed systems and remote execution on heterogeneous distributed systems. We shall revisit many of these key features throughout the rest of the paper.

Demos/MP is a distributed operating system with a migration mechanism for location-transparent reliable interprocess messages [PM83]. A process sends a message to a link. A *link* is a protected global process address accessed via a local name space. A destination host accepts a process and subsequently allocates space for the process state. The transfer of the process state and program (code, data and stack) occurs from source kernel to destination kernel. After the migration, Demos/MP leaves a forwarding address on the source host.

LOCUS provides a Unix-compatible remote execution facility [WPE+83]. LOCUS uses global names consisting of the original host of a process and a local identifier generated by the remote execution site. The original site maintains the names of all processes that are currently executing remotely.

The V system includes process and memory management facilities in the kernel as well as a program manager outside the kernel [TLC85]. The program manager makes the migration policy decision whether to accept a migrating process. The V system uses pre-copying to transfer the state of the migrating process to the destination host. That is, a process continues to execute during the copy of its address space. A page may become invalid after it resides on the destination host. Thus, a destination host may receive several new copies of the same page before the pre-copying is complete. Once the number of dirty pages is small or constant, the source host freezes the process to allow the copy of the remaining dirty pages.

Accent is a distributed computing environment with closely integrated IPC and virtual memory facilities [RR81, Zay87]. In Accent, after a migration occurs, the destination host demand-pages from the source host in a copy-on-reference manner, reducing the time to initially transfer the state. The copy-on-reference mechanism assumes that a process will only access a small fraction of its address space during remote execution.

The Heterogeneous Environment for Remote Execution (THERE) provides a "meta-service" to simplify the adaptation of non-networked, non-heterogeneous applications to a distributed heterogeneous environment [BL88]. In this context, the source and destination hosts are clients and servers, respectively. The THERE programming language supports the construction of execution environments for services, the definition of interfaces for clients to remote services, and system-independent communication between client and server.

Charlotte is a message-based distributed operating system. Each host runs a kernel that handles short-term scheduling and IPC [AF89]. The designers of Charlotte placed the migration policy outside the kernel in a utility. Thus, Charlotte provides the flexibility to easily change the migration policy while keeping it close enough to the kernel for efficient data exchange.

Mermaid extends distributed shared memory to a heterogeneous environment [ZSM90]. Mermaid provides support for converting data to the appropriate type before use by a particular host. The Mermaid project demonstrates the necessities, performance, and limitations involved in data conversion.

The Distributed Library establishes sessions in remote execution environments across a network with various types of machines [Yam90]. Distributed Library differs from standard remote execution facilities by allowing a remote environment to remain established over multiple remote procedure invocations.

The Sprite distributed operating system accommodates process migration in a network file system environment [DO91, DO87]. Sprite uses a low-latency kernel-to-kernel remote procedure call facility. Each process originates on a home node which provides many location-dependent services (e.g., the time of day). When a migration occurs, the source host sends dirty pages to the network file server. After resuming on the destination host, the migrated process demand-pages from the network file server.

Condor provides facilities outside the kernel for checkpointing and process migration [LS92]. Condor is flexible enough to use on a variety of Unix platforms in exchange for the performance penalty of residing outside the kernel.

3 Graph model of single process migration

Current theoretical research couples process migration with load balancing [Hac89]. Queuing theory provides the basis for these models since it is well-suited for modeling load balancing [Kle75]. We introduce a new model which encompasses load balancing as well as other useful purposes for process migration such as remote processing, graceful degradation, and efficient access to resources other than CPU time. Queue-based systems do not allow the analysis of these latter uses of migration.

In our model, we represent a distributed computing environment by a single directed graph. Each vertex in the graph corresponds to an available host and each (directed) edge models both connectivity of hosts and feasibility of tranfer from source host to destination host. Edges and vertices have associated weights. Vertex weights indicate the cost (per unit time) of some resource at the particular host. Edge weights represent the cost of moving a process from source to destination according to the measure described later in this section. The edge weights do not have to satisfy the triangle inequality.[2]

Exactly one vertex in the graph is *active* at a given time. This vertex represents the host where the process currently executes. We consider all other vertices *inactive*.[3] Other models take an approach that is more system-oriented, where all processes in the system are considered in each decision, and all processes are assumed to be part of the load balancing/migration system.

We model changes in the process state by changing the vertex and edge weights appropriately. For example, during execution, a process may enter certain states that are easier to translate to specific hosts, or the process may demand a resource that is not available to the active host (e.g., the process needs to execute on a graphics workstation). In the former case, we lower the edge weights between the vertex on which the process is active and the favorable hosts. In the latter case, we raise the weights of the current host to infinity (forcing migration) and lower the weights of the hosts with the desired resource.

This graph model for single process migration takes into account various policies that are not addressed by traditional models. Process migration policies executed locally at a host make decisions based on the state of the graph for each process on that host. Alternatively, a centralized process migration policy would need to consult a summary of the graphs of the processes on each host before making a decision. In the following sections, we go into more detail about how to determine the vertex and edge weights for the graph model.

The system designer sets the weights of the edges and vertices in the graph by considering the resources in a particular distributed system and the answers to the HOW, WHEN, WHERE, and WHO questions used to determine the appropriate migration policy and mechanism for that system. The answer to the HOW question, or the cost of the migration mechanism, primarily

[2]Indeed, one could superimpose a new set of edges onto the graph that satisfies the triangle inequality, but we wish to allow infinite-weighted edges for the information they contain. Graph reduction is not our concern.

[3]When a process is inactive on a host, the host contains process-specific information (e.g., static memory mappings, library usage, and state translation data), but does not contain the current process state or other dynamic information.

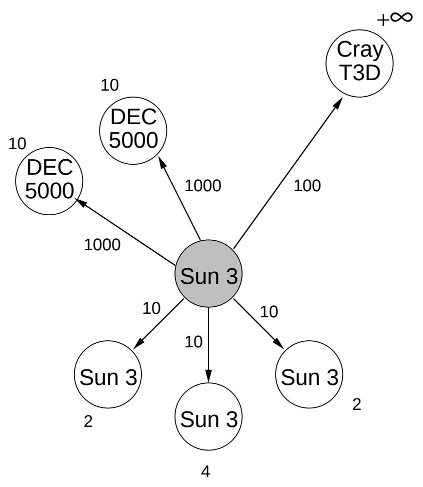


Figure 1: Graph model for single process migration on a heterogeneous distributed system. In this example, we have a network with 4 Sun-3 workstations, 2 DEC 5000 workstations and a Cray T3D supercomputer. The dark-shaded Sun-3 vertex in the center is the host on which the process is currently active. The edge weights are next to the edges and the vertex weights are outside their corresponding vertices. In this case, the Sun-3's and the Cray T3D use the same file system, resulting in lower edge weights, whereas the necessary files would need to be transferred directly to the DEC 5000's, resulting in high edge weights. Unfortunately, to dedicate the supercomputer to applications which perform more efficiently on it, this process cannot migrate to it; thus, the vertex weight on the Cray T3D is positive infinity.

determines the edge weights. The answers to the WHEN, WHERE and WHO questions essentially relate to the migration policy used by a particular distributed system as a means to accomplish a desired effect (e.g., load balancing or real-time response). The migration policy could also affect the edge weights if the policy involves simplifying the mechanism in certain states. For example, if a compiler has finished its preprocessing stage, the only state necessary to continue is in its associated files. Thus, migrating after the preprocessing stage rather than at arbitrary times would decrease the edge weight for migrating the process.

We consider the answers to the HOW, WHEN, WHERE, and WHO questions to derive equations for the edge and vertex weights of the graph. Some of the components of the equations assume constant values for a particular system (e.g., the speed of a uni-processor). Other components change with respect to the state of the system. For example, the size of the allocated virtual memory may differ during the execution of a process which may affect the time to transfer the state.

After initializing the components of the equations for the edge and vertex weights to represent a particular distributed system, a system designer uses the graph model to simulate activity on the target system. Simulations could be used to compare the success of various implementations of migration under certain distributed system environments, goals, and load characteristics. In particular, we would like to determine how effectively the heterogeneous migration facility proposed in this paper performs under certain conditions.

3.1 Edge weights

We first examine the primary components of the migration mechanism. The migration mechanism includes three main components: the negotiation of the migration, the transfer of the process state, and the acquisition of residual information from the source host after migration. The efficiency of the interprocessor communication (IPC) facility is a factor in determining the cost for each of these components. After first examining the cost of the IPC facility, we examine the costs associated with each component of the migration mechanism.

The IPC facility

The speed and reliability of the IPC facility and medium directly affects the cost of many of the components of the migration mechanism. In a multiprocessor, IPC tends to be fast and reliable because of the proximity of the communicating processors. In a local area network (LAN), the distances that a message must travel are longer and the communication medium is slower. A wide area network (WAN) includes hosts which are even further apart than a LAN with an even slower communication medium. The interchange of messages between different subnetworks of the WAN further increases the transit time of a message. In our graph model, we define the *transmission rate* t_{ij} as the speed at which data travels from source vertex i to destination vertex j in bits/sec.

We also consider the complexity of the IPC implementation when evaluating the IPC cost. This consideration assigns costs to the overhead of sending a message which may be quite high in systems with complex IPC facilities such as Accent [RR81]. We define the *message-send overhead* S_i as the cost (in microseconds) of sending a message from vertex i. Similarly, we define the *message-receive overhead* R_i as the cost (in microseconds) of receiving a message at vertex i.

We can now calculate the total cost to send a *message* M_{ij} from a vertex i to a vertex j. If the size of a message is k bits, the *transit time* T for message M_{ij} is

$$\mathrm{T}(M_{ij}, k) = S_i + t_{ij}\, k + R_j \ .$$

Negotiation

Before a process migrates, a negotiation must occur between the source host and the destination host. For our model, we consider the efficiency of this negotiation. The complexity of the negotiation protocol determines the cost of the negotiation. In our model, each vertex i involved in the migration sends n_i *negotiation messages* of small constant size k. The *negotiation cost* N_{ij} of migrating a process between vertex i and vertex j is

$$N_{ij} = \sum_{l=1}^{n_i} T(M_{ij}, k) + \sum_{l=1}^{n_j} T(M_{ji}, k) \ .$$

Transfer of state

The *transfer cost* of the process state typically represents a sizable majority of the cost of migrating a process. The transfer cost includes the removal of the context of a process from the source host, the creation of that context on the destination host, and the transfer of the virtual address space from the source host to the destination host.

The transfer cost should only account for the data (e.g., pages of virtual memory) actually transferred to the destination host. For the copy-on-reference mechanism used in Accent [Zay87], the transfer cost only includes the cost of the pages accessed after the migration. Each transferred page, however, includes a separate overhead cost. For a system such as V [TLC85], a cost may be associated with a page multiple times if that page becomes dirty during pre-copying. The vertex weights, however, would be lower since pre-copying reduces the amount of remaining execution for the process. We also associate additional cost when the logical naming convention of the context on the source host must be mapped to the logical naming convention of the destination host.

We model the transfer cost as a component of the edge weights of our model. We assign a *context-removal cost* CR_i and a *context-creation cost* CC_j for the costs of removing the context from vertex i and creating the context on vertex j, respectively. Since the entire virtual address space may not be transferred, we assign a cost to individual transfers of virtual memory. Thus, the total transfer cost of the migration includes the cost of transferring data accessed at each time t. Since reads and writes may have different costs, when vertex j reads k bits from vertex i at time t, the transfer cost accrues the *read cost* r^t_{ij}. Similarly, when vertex i writes k bits to vertex j at time t, the transfer cost accrues the *write cost* w^t_{ij}.

We determine the total cost for virtual address transfer by summing all the read costs and write costs. The *virtual address read cost* VAR_{ij} for migrating a process from vertex i to vertex j involves all reads performed after the start of the migration. Thus, the virtual address read cost is $VAR_{ij} = \sum_t r^t_{ij}$ for all times t at which a read occurs. We define the *virtual address write cost* VAW_{ij} similarly. Since the virtual address read costs and the virtual address write costs are difficult to predict exactly, a simulator must estimate the total amount of virtual address space that is transferred. With these definitions, the transfer cost from vertex i to vertex j is

$$TRANSFER_{ij} = CR_i + CC_j + VAW_{iq_0} + VAR_{q_1 j}$$

where vertices q_0 and q_1 may represent an intermediate host such as a network file server. Note that all transfer costs among the intermediate nodes are included in the edge weight since no virtual address translation is required at those steps. If the virtual address space transfer occurs directly, then $q_0 = j$ and $q_1 = i$.

The destination host may also need to obtain and compile a copy of the source code in the absence of a local compilation of identical source code on the destination host. For a heterogeneous environment, the edge weights must account for this additional transfer and compilation of code.

Demand translation

In a heterogeneous distributed network, such as a LAN or WAN, the hosts may have different machine architectures. Thus, before communicating data to the destination host, the source host must either translate the data into the format used by the machine architecture of the destination host or translate the data into an external data format. In the latter case, the destination host must translate the data from the external data format into the format used by its machine architecture. We model the *translation rate* x_{ab} as the rate of converting data from the data format of source vertex a to the data format of destination vertex b. When using an external data format c, the translation rate $x_{ab} = x_{ac} + x_{cb}$.

Demand translation could alleviate some of the translation cost of migration by not translating a page until it is needed by the process on the destination machine. The benefits of demand translation are very similar to those of demand paging. For example, when using a network file server, demand translation occurs at each page fault. In a copy-on-reference scheme such as Accent [Zay87], a translation occurs only when the destination host needs a page from the source machine or the network file server. If the migration mechanism transfers the entire virtual address space, pages can remain untranslated on the destination host until referenced. If translation is expensive, demand translation could provide performance improvements comparable to the performance improvements of the copy-on-reference approach to virtual memory access. The total cost of translating the virtual address space from the data format of vertex i to the data format of vertex j in a heterogeneous distributed system is

$$TRANSLATE_{ij} = VAW_{iq_0} * x_{iq_0} + VAR_{q_1 j} * x_{q_1 j}$$

where again vertices q_0 and q_1 may represent a network file server.

Residual dependencies

Leaving residual state of a process on its source machine reduces the immediate cost of migration. If a process must perform certain functions on the source machine [DO87, DO91] or if the source machine is responsible for forwarding messages after the migration completes [PM83], the cost of the migration includes these additional costs. The edge weights should account for the future costs accrued in accessing and maintaining this residual state. The graph model represents this situation by leaving an inactive copy of the process at the source host.

For example, we model the additional *forwarding cost* to forward f_{ij} additional messages from source host i to destination host j for a process as

$$F_{ij} = \sum_{l=1}^{f_{ij}} \mathrm{T}(M_{ij}, \mathrm{length}(l))$$

where length(l) is the length of forwarded message number l.

A number of other residual dependencies may be modeled in a manner similar to modeling the forwarding cost. Some systems use links and must maintain these links between processes after the migration occurs. If the migrated process had been communicating with a large number of other processes, the time to fix the links between these processes could be considerable. Additional time must be spent to collect degenerate links. Some systems require the migrated process to perform certain functions on its source host (e.g., time of day). The edge weights should also count the future costs of communication to perform these functions.

3.2 Vertex weights

Here we examine the costs that determine the vertex weights for our graph model. The vertex weights establish the cost (per unit time) of a resource at a particular host. The answers to the WHEN, WHERE, and WHO questions for process migration factor into the vertex weights.

WHEN

The system designer can use many different criteria for determining when to migrate a process. The system designer sets the migration policy for the system to determine the best candidate processes for migration. The vertex weights change at the moment when the system evaluates the load and selects the candidate processes and the destination hosts for the migrations. If a process knows its resource usage pattern in advance, it can provide the system with hints about when to migrate processes. Our model can take advantage of this knowledge which is not used by queuing theory models. In the compiler example, if the system knows that the preprocessing phase is over, it also knows that a migration can be more easily accommodated at that time.

Migration provides a means of dynamically balancing the load across the hosts of the distributed system. For a given process, the system may determine, at a particular instant in time, that a host cannot provide adequate processing time to a process. The system may detect an under-utilized processor.

A system should use the migration policy to evaluate the system load often enough to balance the load across the hosts of the distributed system. A system may perform this evaluation at fixed time intervals. Critical events occur that may provide better evaluation points for the system load. When these critical events occur, the system should reevaluate the weights of the graph model.

Process creation is a good time to evaluate because, at that time, the system load has changed. If the process provides information about its resource requirements, the system can determine whether that process could produce an imbalance in the expected usage of resources. By migrating the process before execution, it avoids the creation of any new state and the subsequent transfer of the new state.

Another good time to evaluate the vertex weights is when a process moves to a different phase of its execution with new resource requirements. Processes sometimes require real-time response. This phase of process execution requires immediate access to the necessary resources. The evaluation of vertex weights should occur at the instant that a process requires real-time response. The system assigns low values to the hosts that can execute the real-time process and high values to the hosts that cannot accommodate the process.

After a user leaves a host, many processes become idle, freeing up resources for migrating processes [Nic87]. Upon returning to the host, the user may want migrated processes to be evicted. Thus, the vertex weight for that host becomes infinity for each of the migrated processes.

A heterogeneous distributed system provides a large number of different resources each of which may be connected to a specific host [Yam90]. When a process requests a specific resource which is not available on its current host, it must migrate to another host with that resource. At the time that the process needs a specific resource, the vertex weights of all hosts with that resource receive a low value, while weights corresponding to the hosts without that resource become infinite.

The system designer could desire that the system adjust vertex weights at other occurrences of events in the system. If a host has sufficient advance notice of a failure or shutdown, the host can proceed to migrate its resident processes. Processes often require real-time response or exclusive use of its host. The vertex weights change at a rate approximating the urgency of the migration. For example, graceful degradation changes the vertex weights at a constant rate until right before the host fails. At that time, the vertex weights for any other host become very low and the weight of the current host becomes infinite.

WHERE

The system designer establishes a migration policy which determines where to migrate a process. The system designer sets the vertex weights of the graph model by considering the ability of a process to execute on each host and the overall demand on the resources at the host.

A vertex weight signifies the cost (per unit time) of processing at the corresponding host. For load balancing, the system designer uses several considerations in determining the vertex weights: the processing speed of the host, current and anticipated load on the host, and the priority of the process on the host. For distributed systems which contain a homogeneous pool of processors to achieve a high degree of concurrency in the computation, any lightly-loaded host becomes a candidate for receiving a migrateable process [vRvST89, MvRT$^+$90]. Since the processing speed and priority of the process are likely to be the same on each host, the load at a particular host is the primary factor in determining the vertex weight corresponding to that host.

For immediate process evacuation, the system needs to quickly determine the best candidate host to receive the migrating process. Thus, in the absence of time, a detailed analysis cannot be made possibly resulting in a heavily-loaded host receiving migrated processes. The destination host, however, could immediately migrate the process again to another host to balance the load. In these situations, we consider the first processor to accept a migration as the approximate least-loaded machine [TLC85]. Thus, the distance of a host from the current host may be the determining factor in adjusting the vertex weights appropriately.

Distance becomes a more significant factor in a LAN or WAN. The processing cost (per unit time) may increase at a host further away in systems for which the active host must continue to communicate with the original host. We have already accounted for the associated costs of distance between source and destination host in the edge weights.

In some situations, a process may execute more efficiently on a specific host despite a great distance between the source and destination hosts. If a process performs an operation that easily vectorizes, a host with a vector processor becomes a more suitable candidate than other hosts. In a network file system environment, if a process uses very large files, it may execute much more efficiently on the host on which those files reside. The hosts which perform the process more efficiently receive lower vertex weights, which could overshadow large edge weights.

In a heterogenous distributed system, the system migrates a process to a host with the necessary resource. All hosts with the desired resource are the only eligible candidates to receive non-infinite vertex weights. The non-infinite vertex weights are set to approximate the ability of each host to execute the process relative to other hosts.

If all hosts in a distributed system are heavily loaded, the most practical option may be to migrate a process to disk. Thus, we model the disk as a vertex, and move the state of the process to a checkpoint file on disk. In this case, we assign infinity to all vertex weights corresponding to computing processors.

WHO

The decision of WHO initiates the migration affects the costs associated with vertices of the graph model. If a process initiates its own migration, there is no burden on its host to make the decision. If a host, however, enforces the migration policy (e.g., load balancing), that host is responsible for accumulating and analyzing a potentially large amount of information. The processes that execute on that host may execute more slowly due to this extra burden. This overhead would be considered a part of the total overhead (of other processes) running on the host.

The complexity of the policy determines whether it may be a burden on a host. A simple policy such as the first-response policy of the V system [TLC85] requires little additional overhead.

However, a complicated load balancing policy could require a substantial portion of processing time. In this case, a centralized dedicated server relieves the hosts from this overhead. If execution of the migration policy impedes the processes significantly, the vertex weights should be increased for the hosts that perform the analysis for the migration policy decisions.

4 Implementation strategy

In this section, we define the possible levels of implementation in a heterogeneous process migration facility and discuss the primary issues to address when implementing each level. We give four levels of abstraction for implementation. We examine each level of abstraction from the perspectives of checkpointing data and checkpointing code. The goals of this design include no kernel support, minimal daemon, system and root processes, modularity, compatability, and completeness.

The four levels of implementation are the following:

1. Library calls on specific structures. Essentially, the user uses a translation package for specific structures, such as XDR [XDR].
2. Library calls on arbitrary structures. This level of support abstracts the lower-level XDR. All structures are automatically available to be saved. The user must still explicitly invoke the calls to save the structures.
3. Structure tracking. This level of support abstracts the lower-level library. The user registers structures and the migration facility translates them automatically upon migration.
4. Full translation. This level of support translates the stack and automatically registers all structures.

We further divide the four levels of implementation into abstractions of data and code checkpointing for each level. Figure 2 shows the hierarchy among the data levels, D1–D4, and the code levels, C1–C4. Each level uses the services provided by previous levels in the hierarchy.

Each level of data abstraction in checkpointing provides the user with a set of capabilities which determine the data structures that can be checkpointed. We describe how to checkpoint and translate the data structures at the lower levels. The four levels of data abstraction provide the following functions.

D1 No real implementation of system. Primarily, the user must use the interfaces provided by an external data representation interface, such as XDR.

D2 A preprocessor to C strips out specially-labeled structure definitions and creates a library of translation routines. The XDR routines are the default routines for basic structures.

D3 This level of support marks structures and their instances as "migrateable", saving their names and locations in a table. Upon migration, the values in the specified locations are translated automatically.

D4 Everything is migrateable, including the stack. (Optionally, the user can turn off this facility for portions of code.) Stack variables are translated as a group when the current routine calls a subroutine or when a migrate signal occurs. Stack variables can be viewed as named locations that are maintained using the routines provided by level D3.

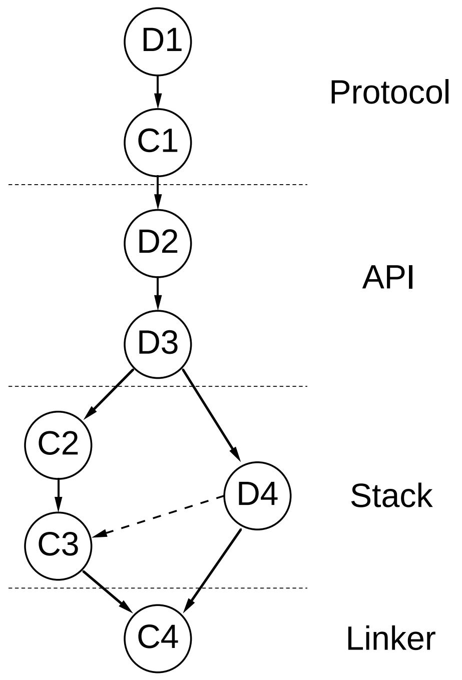


Figure 2: Implementation path for levels of migration of data (D1–D4) and code (C1–C4). The horizontal dashed lines partition the levels into sections. We label each section with the support necessary to implement its corresponding levels.

The code levels of implementation provide the user with the ability to restart a process after migration has occurred. The code levels provide the following functions.

C1 No code translation. The program always starts with a previous state (which could be NULL, i.e., no previous state). This level is useful for testing data level D1. Programs in code level C1 are analogous to single procedures in higher levels.

C2 Left to the user. The user must skip to the appropriate place in a program, reloading structures (stack included), explicitly, as necessary. This level is useful for testing data levels D1 and D2.

C3 Library support. The user can call "auto-skipping" library routines, which optionally restore structures. The user could use support for data level D4 to preserve stack.

C4 Full translation. Automatically skips to the appropriate place. Requires full data translation at specified times (i.e., on Migrate()), at well-defined times (i.e., at procedure calls), and at arbitrary times. This level uses the compiler, linker and the support for data level D4 to bring a program up to where it left off. Code level C4 would require well-defined migration points and routine locations to allow the stack to be fully recreated.

5 The Heterogeneous Migration Facility (HMF)

Heterogeneous distributed computing environments provide a vast amount of computational power which is difficult to utilize by its very nature. The Heterogeneous Migration Facility (HMF) hides the details of communication and translation between two disparate host architectures and separates the migration and checkpointing functionality from the main function of an application program.

The design of HMF is based on the graph model for single process migration and the implementation strategy described in Sections 3 and 4. The model separates process migration into four distinct components: *how* a process migrates, *where* a process migrates, *when* a process migrates, and *who* controls the decisions of the first three components. The lowest layer of HMF implements the *how* component by providing a format and protocol for a program's state. Higher layers of HMF contain the *where* and *when* components, and the user currently determines the *who* component.

The main purpose of HMF is to provide a simple and flexible mechanism for users to introduce process migration into computationally intensive programs. Such programs, which may run for days at a time, are particularly vulnerable to load imbalance and system failure. While standard definitions of process migration do not include checkpointing, we have included it in HMF for two reasons: checkpointing fits into the model (and thus should fit into any implementation based on the model), and checkpointing combats system failure (although to a smaller extent than traditional migration). When we refer to migration, we include checkpointing; in the model, a checkpointed process has been "migrated to disk" (the *where* component has made this decision).

5.1 Overview of HMF

The current implementation of HMF includes data migration only. The user must provide mechanisms for code migration. Our example program later demonstrates one possible way to handle code migration.

The layered design on HMF provides for easier introductions of new features, bug fixes, and optimizations, and gives the user more flexibility in choosing the proper level of control over migration. HMF provides the following layers in its design.

```
      auto
 check migrate
   translate
    objects
      xdr
```
files, streams, & memory

The `objects` library forms the core of HMF. Routines in this library manipulate *objects*, which fully describe data structures, control variables, and I/O streams (in a future revision). Each object has a set of attributes, including *name*, *type*, *data*, and *format*. These basic attributes are used by the higher layers of HMF for data location and translation. Other attributes may be added by the user. (Note: HMF can translate any object that contains these four attributes, in the proper format.) The `objects` library also provides the low-level input and output of objects to files, streams, and memory, and gives the user an easy way to migrate to a remote host.

Translation of HMF objects from one host architecture to another is handled by the `translate` library. This collection of routines converts data from the object representation to the internal representation of the host, preserving type and proper alignment of all fields in structures, arrays, and unions. Future implementations will allow greater flexibility in adding custom translation modules for specialized data structures, but the basic library can translate any C structure.

The `check` and `migrate` libraries form the first layer of *user routines*. They serve to simplify the interface to HMF, by hiding the details of objects and translation.

Currently, the highest layer of HMF is the `auto` library. This set of routines provides a common, signal-driven mechanism to automatically migrate a process state (as described by the user). The interface to this library is described below.

5.2 HMF type descriptions

HMF preserves complex data structures across migrations, with minimal interaction from the user. For each structure that is saved in the state, HMF maintains a *type description*, which is actually an array of type `int`. Type descriptions fully define the structure of a block of data, and are used to pack and unpack the data during migration. HMF includes a preprocessor that automatically generates type descriptions of type definitions designated by the keyword `checktype` as below.

`checktype typedef` *type-definition*

HMF builds type descriptions up from a set of predefined *type numbers*, just as structure definitions (in C) are built from a set of predefined types. All basic types (e.g., `int`, `char`, and `float`) have corresponding type numbers as specified in Figure 3. We translate the simple type `int` into the following type description.

```
{ typeINT }
```

Type descriptions may also contain *structure numbers* which help HMF parse the description. We define structure numbers for structures such as arrays, pointers, `struct`s, and `union`s. The following example shows a type definition for an array of `int`s of constant length `MAX_IN_ARRAY` defined by the user.

```
{ typeARRAY, MAX_IN_ARRAY, typeINT }
```

Similarly, a pointer to an `int` is the following type description.

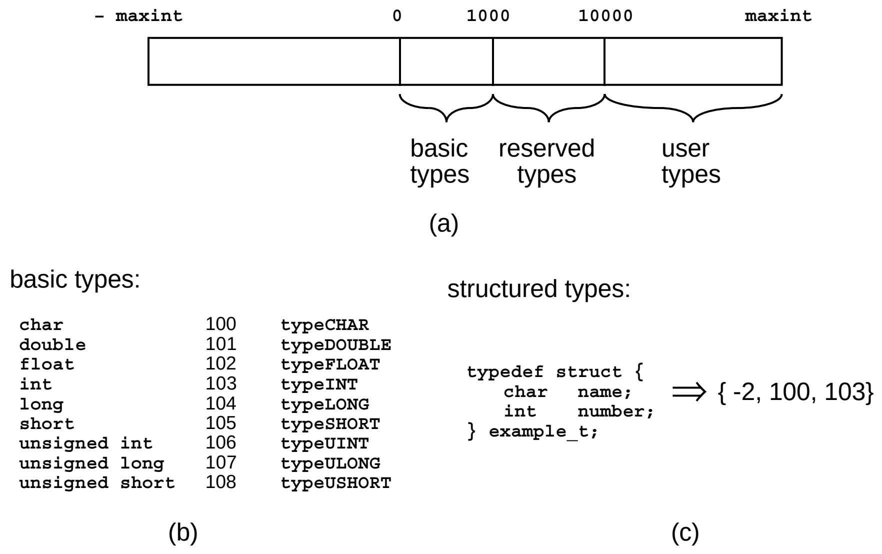
```
{ typePOINTER, typeINT }
```

Of course, instead of building type descriptions from basic type numbers, any type description can include other previously-defined type descriptions. The following example is a type description for a pointer to an array of `MAX_IN_ARRAY` `int`s.

```
{ typePOINTER, typeARRAY, MAX_IN_ARRAY, typeINT }
```

The type descriptions for `struct`s are a bit more complicated, because the type description must also specify the number of fields in the structure. For example, the C structure

```
struct example1_s {
    int code;
    char key;
    int key_id;
```



char	100	typeCHAR
double	101	typeDOUBLE
float	102	typeFLOAT
int	103	typeINT
long	104	typeLONG
short	105	typeSHORT
unsigned int	106	typeUINT
unsigned long	107	typeULONG
unsigned short	108	typeUSHORT

```
typedef struct {
   char   name;
   int    number;
} example_t;
```

$\Longrightarrow$ { -2, 100, 103}

Figure 3: External type numbers. The user designates a type for which to create a type description by preceding the type definition by the keyword `checktype`. Each `checktype` receives a type number. We show the ranges of type numbers used for each type in (a). The basic types (e.g., `int`) have a type number in the range 0–999. We list the type numbers for the most basic types in (b). Type numbers 1000–9999 are reserved for types such as `typeARRAY` and `typePOINTER`. The HMF preprocessor or the user assigns a number between 10000 and `maxint` to a user-defined type such as `example_t` in (c).

```
    };
```

translates to the type description

```
    { -3, typeINT, typeCHAR, typeINT } .
```

Figure 4 shows the translation of an even more complex `struct`. Unions are also represented by the reserved type `typeUNION`.

5.3 HMF `auto` library

The `auto` library is currently the highest interface available to users of HMF. Through the `auto` interface, the user registers typed data structures for inclusion in future state descriptions. The user then calls an idle routine every so often, which handles any requests for migration. Other routines initialize the library and read in any previously existing state.

The main goal of the `auto` library is to provide an automatic migration mechanism. Once a data structure has been "registered" with the library, it becomes part of the process state, and will be migrated when necessary. This mechanism relieves the user from the task of handling migration requests.

Before using any routine in the `auto` library, the user must first call

```
    auto_init(argv[0]);
```

```
struct example2_s {                            { typePOINTER,
   int uid;                                      -4,
   char *name;                                   typeINT,
   char passwd[8];                    ⟶          typePOINTER, typeCSTRING,
   int gids[MAX_GROUPS];                         typeFSTRING, 8,
};                                               typeARRAY, MAX_GROUPS, typeINT }
typedef struct example2_s * example2_p;
```

Figure 4: Translation of a pointer to a complex data structure to an HMF type description. The type description begins with `typePOINTER` to signify that the type is a pointer. Since the following number is negative (-4), the data structure is a `struct` with 4 fields. The type `typeCSTRING` has no following parameters and type `typeFSTRING` assumes that the next parameter indicates the length of the string.

to initialize the internal structures, where `argv[0]` is the name of the program. HMF uses the name of the program to create a unique state file (if the process is migrated to disk), or to start the proper program on a remote machine (if the process is migrated to a remote machine). When finished with the `auto` library, the user calls the `auto_done` routine to free any private memory used by `auto` and unregister all previously registered data structures.

The user registers data structures by calling the `auto_track` or `auto_register` functions. For example, to register a variable that contains a loop counter `i` of type `int`, the user declares the type description

```
int type_list_int[] = { typeINT };
```

as described above. The following call to `auto_track` registers the variable `i`.

```
auto_track(``i'', &i, sizeof(int), type_list_int, 1);
```

The `auto_track` function merely allocates a *tracking structure* of type `var_elem_t`, places the appropritate values in its fields, and calls the `auto_register` routine. The user can call `auto_register` directly as follows.

```
var_elem_t ve;
ve.v_name = ``i'';
ve.v_data = &i;
ve.v_data_len = sizeof(int);
ve.v_type_list = type_list_int;
ve.v_list_len = 1;
auto_register(&ve);
```

Data can be removed from the state, or unregistered, by calling either the `auto_untrack` or `auto_remove` routine. To be consistent, after calling the `auto_track` routine to register a data structure, the user should call the `auto_untrack` routine to unregister it. For example, the user would call

```
auto_untrack(&i);
```

to unregister the variable `i`. This routine calls the `auto_remove` routine to unregister the variable and then frees the memory used for the tracking structure of type `var_elem_t`. The user calls the auto_remove routine to unregister the data directly by passing the address of its var_elem_t by calling

```
auto_remove(&ve, NULL);
```

or by passing the address of the data structure to unregister it as in the following call.

```
auto_remove(NULL, &i);
```

Future revisions will allow you to unregister by name.

In order for the auto library to automatically migrate your process, the user must periodically call auto_poll to check internal flags and migrate the process if necessary. The user can disable checkpointing or migration by passing a mask to auto_poll. (Note: all migrations occurs within this routine.) The user checks the return value to learn whether the program has been checkpointed or migrated. For example, if only checkpointing the program, the user performs the following poll on a periodic basis:

```
if (auto_poll(AUTO_CHKP) == AUTO_CHKP) {
/* process has been checkpointed */
}
```

Normally, the user continues execution after checkpointing the process.

If the user wants to allow the process to be migrated, the following poll determines if the migration has been successful.

```
if (auto_poll(AUTO_MIGR) == AUTO_MIGR) {
/* process has been migrated */
}
```

If the migration was successful, the process should stop execution on its current host and exit from the program, because the process now executes on the remote host.

To enable both checkpointing and migration, the user polls in the following way.

```
if (auto_poll(AUTO_ALL) & AUTO_MIGR) {
/* process has been migrated, stop execution */
}
```

In this example, if the process has been migrated, execution should be stopped, otherwise the process should continue. In all cases, auto_poll returns 0 (zero) if the process has not been checkpointed or migrated.

The auto library also includes a routine to initialize state to previous values if the process has been checkpointed or migrated. The user calls the auto_read routine after registering any variables. This routine initializes the registered variables to the values found in any existing state which either resides in a checkpoint file or arrives in a message from a process on the source host.) The return value for the auto_read routine indicates whether or not there was a previous state. Below is an example of an invocation of auto_read.

```
if (auto_read(AUTO_ALL)) {
/* there was a previous state, state is initialized */
} else {
/* no previous state, set state to initial values */
}
```

Appendix A shows a sample program which uses all the features described in this section.

We have also rewritten the password cracker of [Bis88] using HMF. The password cracker is a good test program for HMF because it includes one big loop which provides a natural checkpoint location after each iteration, and the program runs a long time without any user interaction.

6 Conclusions

We have examined the feasibility of process migration in a heterogeneous distributed system. A system designer can use our graph model to study the possible benefits of migration for a wide range of system goals which include performance, reliability, and availability. The Heterogeneous Migration Facility (HMF) demonstrates the support that is necessary for data migration in an application programmer interface (API).

Benefits and drawbacks of the graph model and HMF

The graph model provides us with an abstraction of the resources in a heterogeneous distributed system. In order to fit a given system into the model, there needs to be some way to represent the various resources. The system designer can use the graph model to analyze the system by classifying the available resources and assigning weights to the graph model appropriately. In this way, the graph model serves as the foundation for the implementation of a migration/load balancing system like HMF.

The graph model accounts for all factors contributing to migration/load balancing costs. It also views the system from a single process view. Other queuing theory models assume all processes in a system are part of the load balancing mechanism, which is usually not the case. Our model still accommodates the system-wide approach used in queueing-based models.

The major drawback of the graph model is the complex analysis resulting from the many factors in measuring the costs of process migration. Simulation will determine the difficulties involved in using the model for accurate analysis.

HMF has several advantages. HMF provides information to the load balancing mechanism. HMF is operating system independent (which is important in a heterogeneous environment). HMF allows multiple "entry levels" for the user.

Unfortunately, levels D1–D3 require application programmer support. Thus, each program must maintain details about its own state. Additionally, in its current implementation, HMF does not handle register-based variables efficiently. Such optimizations would be handled as part of the D4 implementation.

A simulator based on the graph model would allow a system designer to study the success of various migration policies and mechanisms as well as system parameters (such as network speed) in meeting the needs of the target user community. How accurately can a simulator based on the graph model predict real systems behaviour? Are there other factors that should be considered in determining the edge and vertex weights for the graph model?

HMF can be further extended to address the issues of full translation and automatic migration of code including the stack. Is HMF the best approach for providing process migration in heterogeneous process migration? Is the recent work on transportable intelligent agents (TIA) flexible enough to address all the scenarios in which migration provides benefit?

References

[AF89] Yeshayahu Artsy and Raphael Finkel. Designing a process migration facility: The Charlotte experience. *Computer*, 22(9):47–56, September 1989.

[Bis88] Matt Bishop. An application of a fast Data Encryption Standard implementation. *Computing Systems*, 3(1):221–254, Summer 1988.

[BL88] Brian N. Bershad and Henry M. Levy. A remote computation facility for a heterogeneous environment. *Computer*, 21(5):50–60, May 1988.

[DO87] Fred Douglis and John Ousterhout. Process migration in the Sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, September 1987.

[DO91] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software-Practice and Experience*, 21(8):757–785, August 1991.

[Hac89] Anna Hac. A distributed algorithm for performance improvement through file replication, file migration, and process migration. *IEEE Transactions on Software Engineering*, 15(11):1459–1470, November 1989.

[Kle75] Leonard Kleinrock. *Queuing Systems Volume I: Theory.* John Wiley & Sons, New York, 1975.

[LS92] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the Unix kernel. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 283–290, January 1992.

[MvRT$^+$90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[NBL$^+$88] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31(3):258–273, March 1988.

[Nic87] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium of Operating System Principles*, pages 5–12, November 1987.

[PM83] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium of Operating System Principles*, pages 110–119, 1983.

[RR81] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, pages 64–75, 1981.

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the Tenth ACM Symposium of Operating System Principles*, pages 2–12, December 1985.

[vRvST89] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. Performance of the Amoeba distributed operating system. *Software – Practice and Experience*, 19(3):223–234, March 1989.

[WM85] Yung-Terng Wang and Robert J.T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, 34(3):204–217, March 1985.

[WPE$^+$83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 49–70, October 1983.

[XDR] *External Data Representation Standard: Protocol Specification.* RFC1050, ARPA Network Information Center.

[Yam90] Michael J. Yamasaki. Distributed Library. Technical Report RNR-90-008, NAS Systems Division, NASA Ames Research Center, April 1990.

[Zay87] Edward R. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium of Operating System Principles*, pages 13–22, November 1987.

[ZSM90] Songnian Zhou, Michael Stumm, and Tim McInerney. Extending distributed shared memory to heterogeneous environments. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 30–37, 1990.

A HMF sample code

The code in Figure 5 is a sample program using the application programmer interface (API) of the Heterogeneous Migration Facility (HMF). The file `typemap.h` contains the declarations of the type numbers for the basic and reserved types (e.g., `typeINT`). The file `auto.h` includes the prototypes for the HMF routines.

After initially declaring some variables, the program initializes the `auto` library with the `auto_init` call. We use the `auto_register` routine to register the variable `anint` which we name "`my var`" to be checkpointed. The `auto_read` call checks for previous state and initializes the variable `anint` if none exists. Inside the `for`-loop, we check to see if the host has migrated this process to another processor by calling the `auto_poll` routine. We continue to increment the variable `anint` until the process has migrated. Once the process migrates, the program breaks out of the `for`-loop, unregisters the variable `anint` by calling `auto_remove`, and exits the `auto` library by calling `auto_done`.

```
#include ``typemap.h''
#include ``auto.h''

main (argc, argv)
   int argc;
   char *argv[];
{
   var_elem_t ve;
   char vn[10];
   int anint;
   int itype = typeINT;

   strcpy(vn, ``my var'');

   auto_init(argv[0]);

   ve.v_name = vn;
   ve.v_data = (char *)&anint;
   ve.v_data_len = sizeof(int);
   ve.v_type_list = &itype;
   ve.v_list_len = 1;
   auto_register(&ve);

   if (!auto_read(AUTO_ALL))
      anint = 0;

   for (;;) {
      if (auto_poll(AUTO_ALL) & AUTO_MIGR)
         break;
      anint++;
   }

   auto_remove(&ve, NULL);
   auto_done();
}
```

Figure 5: A sample HMF program.