Process Migration for Load Balancing and Fault Tolerance

Asad Hamid, Khanh Nguyen, Paul Nguyen, Simon Turner

Department of Computer Science and Engineering
University of California, San Diego

Abstract

Process migration provides a means for improving load balancing and fault tolerance in distributed computer systems by controlling the distribution of processes among the different hosts. This paper surveys research that has been done in the areas of policy, mechanism, and implementation of process migration for the purposes of load balancing and fault tolerance.

1 Introduction

Local networks of workstations are extremely popular and offer potential advantages over larger mainframe systems in the areas of scalability, redundancy and cost. One important advantage offered by these types of distributed computer systems is a substantial pool of available resources, including CPU, virtual memory, secondary storage, and machine specific hardware resources. The resource availability offered by these systems can be exploited by providing a means for migrating processes among the different workstations in the network, rather than limiting processes to run on the workstation on which they are created. This allows machines that would otherwise be idle to be used for useful work, and also allows processes to be collocated with resources or data that may have otherwise been unreachable or reachable at the cost of a significant amount of network bandwidth.

Process migration involves the transfer of a process from a local host to a remote host. A process can be migrated to a remote host either at the time of process creation or during process execution. A *process migration policy* is required to define the rules for process migration regarding what, where and when a process is to be migrated. A *process migration mechanism* is required to manage the actual transfer of a process, including all required process state, from one machine to another in an efficient and reliable manner.

Potential advantages that can be realized by a distributed system that offers process migration include the following:

- Load balancing – processes can be assigned to idle machines to exploit the availability of system resources on all machines in a network.
- Fault tolerance – redundant processes can be assigned to multiple machines to improve the probability of completion in the event of a failure.
- Reliability – processes can be migrated to other machines before a machine is taken down for maintenance.
- Concurrency – applications can be designed to exploit process migration to maximize concurrency (e.g., Sprite pmake [9])
- Availability – similar resources can be made available on multiple machines.
- Reduced network traffic – processes can be migrated to machines that offer services that may induce significant network traffic for remote access.
- Remote resource access – processes can be migrated to machines that offer resources that are not accessible remotely.

Potential disadvantages and/or limitations of systems that offer process migration include the following:

- Additional complexity – process migration policies and mechanisms can add non-trivial complexity to a distributed system, depending on the scope of the requirements.
- Additional overhead – additional processing is required to execute process migration policy and mechanism.
- Heterogeneity – process migration between heterogeneous machines is problematic due to differences in instruction sets and processor architectures.
- Portability – kernel level implementations of process migration can restrict portability and require that all machines in a network are running the same operating system.

This paper is organized into six sections. In the next section, policy issues regarding process migration are discussed. In section 3, an overview of process migration mechanisms is provided. Section 4 summarizes issues that need to be addressed in the implementation of process migration on a distributed system. Section 5 addresses the topic of system load metrics for use with process migration policies. Section 6 presents conclusions and suggestions for possible directions of future research in this area.

2 Policy

Process migration policy directs the migration of processes to particular machines in a distributed system to maintain a desired level of resource availability and fault tolerance. Process migration policy can be executed either automatically by system software, or manually by individual users.

Process migration policy involves the identification of candidates in the following three categories, and the distribution of processes in a way that best meets the requirements for resource availability and fault tolerance in a particular system:

- Machines from which to migrate processes,
- Machines to which to migrate processes,
- Processes that are eligible for migration.

Candidate source and destination machines are identified by comparing current resource availability to established criteria. Candidate processes for migration are identified by comparing process resource usage patterns to established criteria.

Process migration policy can be applied statically when processes are initiated and also dynamically while processes are executing. The former will be referred to as static initiation policy and the latter will be referred to as dynamic migration policy.

The process migration policy for a system should be selected based on the goals and characteristics of that particular system. Static process initiation policies are generally simpler to implement due to their static nature, and are well suited to systems that are characterized by many short-lived processes. Dynamic process migration policies are generally more complex to implement due to their dynamic nature, and are better suited to systems that are characterized by long-lived processes whose resource requirements may change over time, or whose computations must be more reliable due to the larger cost of repetition in the event of failure.

Static initiation policy

At the time a process is initiated, the processor on which to initiate the process can be selected in a way that meets the system requirements for resource availability and fault tolerance. Fault tolerance requirements may also dictate that a particular process be assigned to multiple processors to improve the probability that a computation will be completed in the event of a system failure.

Process initiation policy for fault tolerance has been investigated by Friedman and Mosse in their examination of fault tolerance and load balancing in distributed, replicated servers [1]. In this research, a distributed server was constructed from a cluster of workstations. Some of the nodes in the cluster acted as front end nodes which accepted requests from external clients and assigned those requests to other nodes in the cluster for processing. The rest of the nodes served as query elements which processed requests and returned the results back to the front end nodes. Process assignment policies based on random assignment, round-robin, least pending requests, lowest CPU load were evaluated. The results showed that the fault tolerance scheme affected the load balancing, and that the load balancing scheme affected the fault tolerance. It was concluded that choosing the right fault tolerance and load balancing schemes is critical to designing successful reliable distributed servers, and that simplicity was an important factor in selecting a policy that could be successfully implemented.

In the research carried out on the Amoeba system [6], a system service called the load balancer selected which processor would be assigned to run a process at the time each process was initiated. The load balancer was a centralized server that managed a processor pool and assigned a process to a processor according to the configuration of the system. To make a process allocation decision, the Amoeba load balancer examined the available memory, memory fragmentation and whether the text and data segments are cached in memory. The following formula was used to determine which processor will run a process:

$$\text{Value (h, hd)} = \frac{\text{cpuspeed}_h + \text{memory (h, pd)}}{\text{runnable}_h}$$

where h represents a particular processor, pd represents a particular process, cpuspeed_h represents the speed of a processor, runnable_h represents the average number of runnable threads and memory(h,pd) represents the amount of available memory processor h has available to execute process pd. The load balancer selected a processor which had the greatest resulting value for process pd. If there was more than one processor with the same greatest value, the load balancer randomly selected a processor from this set of processors.

In the implementation of process assignment in the Sprite operating system [9], the selection of idle hosts was fully automated but the other policy decisions were only partially automated. The rationale for this decision was based on the fact that the system was composed of personal workstations where users were content to run most of their processes on the local host, but also expected to have complete control of their workstations. To implement idle host selection, each workstation ran a background process called a *load-average daemon* which monitored the usage of that workstation. When the workstation appeared to be idle, the load-average daemon notified the *central migration server* that the workstation was ready to accept migrated processes. The Sprite system provided two forms of user controlled process assignment: pmake and mig. The *pmake* utility provided the same services as the standard make utility, except compilation was parallelized as much as possible by assigning as many parallel compilations to idle workstations as possible. The *mig* utility allowed users to execute shell commands on idle hosts and could also be used by users to migrate executing processes from the local workstation to an idle workstation. One other form of process migration that was provided was *eviction*. The load-average daemon running on a workstation could detect when a user started using that workstation by detecting keystrokes or mouse motion. Upon detecting the presence of a user, all foreign processes would be evicted and migrated back to their home machines, from which they could be re-migrated if desired.

Initiation policy is attractive due to its simplicity. The system resources are balanced at the time of process creation. This approach does not add more processes to overloaded processors, but cannot reduce the load on processors that may become overloaded regarding a particular local resource.

Dynamic migration policy

During execution on a distributed system, it may be desirable to migrate a process from one machine to another for a number of different reasons. If a machine in the system is going to be taken down for maintenance, all processes running on that machine can be migrated to other machines before the particular machine is taken off line. In the event of a system failure, it may be possible to migrate processes to other machines before the local machine completely shuts down. If a particular process needs to access a particular hardware resource or data on a remote host, it may be more efficient to migrate the process to the resource, rather than accessing the resource remotely.

In the research carried out on the Amoeba system [6], the load balancer service periodically checked the latest workload information to determine whether a processor was overloaded. In the event that a processor became overloaded according to the particular criteria used, processes were reallocated to other processors to restore the system resource balance.

To achieve optimal system performance, it is important to consider the overhead required to allocate and migrate a process relative to the running time of a process, the overhead required to make a migration policy decision, and the time required to wait for migration relative to the running time for a process. It is reasonable to move a process that has a long remaining executing time to a different processor if the time required to migrate the process is less than the time that will saved by migrating to a processor that can dedicate more CPU time to the process. A computationally intensive process can also be a good candidate for migration if the time required to migrate the process is less than the time that will be saved in process execution time. A process with small size implies less time required to migrate which is also an important factor in selecting processes for migration.

The Amoeba system implements three heuristic methods to choose a process for migration:

Most CPU intensive process. A process with CPU-intensive will be selected for migration.

Longest executing process. According to the Markovian property, a process with the longest execution time is predicted to have the longest remaining time.

Execution time and CPU usage. A combination of the previous two policies can also be used to select processes for migration.

Additional information that can be used for process selection includes weighted sum of CPU consumption and execution time.

To determine which processors are overloaded and which are under-loaded, the Amoeba load manager took into consideration the number of processes on a CPU and the CPU utilization. The Amoeba system used a static threshold value to determine the maximum number of processes and maximum utilization on a processor. A processor that exceeded these thresholds was considered overloaded. Processes running on this processor were migrated to the processor that had the lowest value in these thresholds.

The specific migration policies used by the Amoeba system to identify hosts that could accept migrating processes were the following:

Queue length. In the Queue length policy, the number of processes is the primary factor in identifying over-loaded and under-loaded processors.

CPU utilization. In contrast to the Queue length policy, the CPU utilization is the primary factor in determining over-loaded and under-loaded processors.

Queue length and CPU utilization. The preceding two policies can be combined to consider both the number of processes and the CPU utilization in identifying under-loaded and over-loaded processors.

Dynamic process migration is attractive due to the ability to maintain load balancing dynamically and the ability to move processes to meet changing resource requirements. However, dynamic process migration policies are more complicated than static user initiated policies, and are therefore more complicated to implement and verify.

3 Mechanism

Once the decision to migrate a process has been made, a mechanism for performing the migration is needed, that is, what must be done to move a running process to a different host and continue execution there. Migration involves stopping the execution of the process in its current host, saving the process state, transferring the state into the destination host, and restarting the process on the destination host. Many alternative methods have been proposed to perform these tasks. Each alternative satisfies a different set of constraints regarding the environment, performance, interoperability, etc. We will first look at how to save and recover the state of a process, which is commonly referred to as checkpointing and restarting. We will then look at the different process migration algorithms and their relative strengths and weaknesses.

Checkpointing and restarting

Saving the state of a process is a very challenging task. The run-time state of a process includes the process static memory (the global variables), allocated virtual memory on the heap, register values, subroutine invocation stack, file descriptor table, communication connections, locks, etc. Some of these are easy to save, but others have many inherent complications. For example, the external state entities may include interactions with other processes, and it is very difficult to capture the transmitting messages, opened locks, etc. The problem is further complicated when dealing with a heterogeneous distributed system instead of a homogeneous one, that is, when the destination host does not run on the same platform and architecture. For example, let's look at the case when the distributed system is a network of all Unix hosts. Capturing the state of a process is as easy as getting the run-time information from the Unix kernel. Once the state arrives at the destination host, which is also running Unix, no translation is needed because the instruction sets, data representation, memory addressing, and address space structure are all the same. However, in a heterogeneous system, no assumption can be made about these environment-dependent factors. The semantics of the state entities must be somehow translated.

One simple mechanism is to utilize the object serialization offered by interpreted languages such as Java, and let the interpreter handle the job. This method, however, has a tremendous setback in performance, which is the central problem that load balancing and migration tries to solve. Two other current mechanisms proposed for checkpointing are Process Inspection and Process Hijacking. Process Inspection focuses on the platform independence issue, while Process Hijacking tries to solve the problem of requiring the migrating process to re-link with a checkpoint library.

Process introspection. The Process Introspection approach to checkpointing, as described by Ferrari et.al. [10], modifies the program code of the migrating process so that it can save its own state. This modification is done at the intermediate code (such as the .o files in Unix), rather than the low-level native code or the high-level code. This choice avoids the problem of inserting into low-level code which is not platform independent, and of inserting into high-level code which requires programmers to be aware of the introspection process. Insertions to the code include a global variable called *CheckpointStatus*, a new subroutine called P_{chkpnt}, poll points, and other instructions to maintain a table that contains dynamically allocated state entities. The *CheckpointStatus* variable indicates whether a checkpoint has been requested or the checkpointing process is being carried out. The duty of the P_{chkpnt} subroutine is to output the values of all static and dynamic state entities, some of which are maintained in the table above. A *poll point* is an instruction that checks for a checkpoint request via the *CheckpointStatus* variable. A poll point instruction is added after each subprogram activation instruction. Optionally, a poll point can also be added between any two instructions if more frequent checking is desired.

Consider a scenario that illustrates how these modifications facilitate checkpointing. The *CheckpointStatus* variable initially indicates that no checkpointing is in progress. All dynamic allocations and deletions are maintained in a table throughout the execution of the process. When a checkpoint is requested, an interrupt notifies the handler code to set the *CheckpointStatus* variable to indicate this request. When the execution reaches a poll point, the *CheckpointStatus* variable is examined, and thus the process starts saving its state. First, the process saves the current subprogram activation and other local state entities. Second, the process outputs a marker indicating the logical location in control flow. Third, the subprogram must return to its caller, where these three steps are recursively applied to the caller. If the current subprogram is the outer-most, the P_{chkpnt} subroutine is also executed to save the global variables and dynamically allocated variables.

Corresponding modification to the original process's intermediate code must also be made to enable restarting, which include allowing the CheckpointStatus variable to indicate whether a restart is in progress, and adding a new subroutine called $P_{restart}$ that read the saved entities from P_{chkpnt} and re-initiate them. The activation stack is recovered by recursively look at the saved markers of each subprogram and re-constructing the its activation record and other saved information. Also note that some external state information, such as locks and communication connections, cannot be saved in this process. This mechanism relies on some form of support of the process's environment to checkpoint these entities. A more difficult alternative method would be to insert the ability to save external state into the process itself.

Process Introspection thus solves the problem of checkpointing in a heterogeneous distributed system because insertions are made to the intermediate code. The insertions do degrade performance, but only to an acceptable level, as measured by experiments [10]. The biggest setback about this approach is the requirement to re-link the intermediate code after modifying it. This problem can hinder the resource manager in its load balancing operation, since many processes are proprietary, and their intermediate codes are not available for modification.

Process hijacking. The newer Process Hijacking approach, as described by Livny et.al. [11], tackles the biggest problem with the Process Introspection approach which is the requirement to re-link the intermediate code. Similar to Process Introspection, Process Hijacking supports process migration by adding a checkpointing capability into the migrating process, but it does so in a dynamic way. This is possible by utilizing a run-time code re-writing library, which is normally used for monitoring, debugging, and profiling running process. One such library is the DynInst API, which provides an architecture-independent interface to splice code patches into a running process. Thus instructions to allow the migrating process to save its state can be added to any process, even if the process is running, allowing it to perform checkpointing by itself.

One other issue that Process Hijacking tackles gracefully is the difficult problem of saving external state, such as open files. Rather than trying to save the external state and move that state into the destination host, Process Hijacking allows the migrated process to refer back to the source host to access those state entities. This means the external state still resides on the source host, inside a *shadow process*, which is forked by the migrating process just before it is terminated. By default, on most environments the forked child inherits all file descriptors from its parent, thus preserving the parent's external state information. The shadow process continues to live after the migration and communicates with the migrated process when needed. This communication is done by making remote procedure calls (RPC) from the migrated process back to the shadow host. As a result, all instructions that access these external state information, (i.e. the system calls) must be changed into remote procedure calls. The same library above can facilitate this change by making modifications to the process before migrating.

By tackling the above two issues, Process Hijacking successfully deals with the weak points of Process Introspection regarding the requirements to re-link and save external state. Measurements show that this advantage does come with a slightly higher cost in the performance. Having a shadow process at the source host also reduces reliability when the source host crashes or when the network is down, and puts a small workload on the source host.

Migration algorithms

The two migration mechanisms above, Process Introspection and Process Hijacking, are examples of two different general classes of migration algorithms, which are Eager Copy and Lazy Copy, respectively. These are not the only two classes of migration algorithms available. There exist other alternatives that each have their own advantages and disadvantages. The five recognized classes of migration algorithms are Eager Copy, Lazy Copy, Pre-Copy, Post-Copy, and Flushing. These algorithms differ primarily in the following three major issues:

- How much of the process' state is transferred form the source host to the destination host?
- When is the execution of the process on the source host suspended?
- When is execution of the process on the destination host resumed?

Eager Copy. Eager Copy is the most straightforward and probably the most popular mechanism for process migration. After the decision to migrate a process has been made, the process execution is immediately suspended on the source host. Then the complete state of the process is captured and transferred to the destination host. Only after all state transfer has been completed, can the process resume execution on the destination host. The main advantage of this algorithm is its simplicity, but the delay between process suspension and process resumption is generally the longest of the different classes of process migration.

Lazy Copy. Lazy Copy is the complete opposite of Eager Copy. This approach resumes execution on the destination host as soon as possible. Only the minimum state information needed to re-start process execution is transferred to the destination host. Execution is resumed immediately after this partial data arrives on the destination host. All other state information is transferred later based on demand. This means that when the migrated process needs to obtain some state information that it does not already have, it must make a request to the source host. This request is trapped to the kernel just like any other system call, except that the delay may be longer because a network connection to the source host is needed to get the required data. This algorithm is superb due to a very small delay between the suspension and resumption of the migrating process. It can also reduce the overall network traffic, because some state information might never be requested. The major disadvantage is the *residual dependency* problem. The state of a process is now contained in two different hosts.

Pre-Copy. Pre-Copy takes advantage of the fact that the execution of a migrating process and the transfer of state information can take place in parallel. When a migration request is made, the pre-copy mechanism starts transferring state data immediately, but does not stop execution on the source host. Execution is only suspended after the state transfer is complete. At this time, a final flush of the address space to the destination host must be performed. This is because some pages may have changed during the first state transfer. Only after this flush is done can execution resume at the destination host. Time lag between the suspension and resumption is reduced to the time to transfer the final flush, which should be significantly smaller than the lag for Eager Copy. The overall lag, however, is longer than Eager Copy because some information has to be transferred twice. Since the process is only resumed after all updated state information has arrived at the destination host, no residual dependency exists. This method might not be a preferred choice for load balancing because the process continues to use the source host's computing resources for a while before releasing them.

Post-Copy. The newer Post-Copy algorithm, as introduced by Richmond and Hitchens [3], is similar to Pre-Copy except that it eliminates some of Pre-Copy's problems, while not introducing other major problems of its own. Like Pre-Copy, Post-Copy allows a process to execute in parallel with the transfer of state data. The difference is that execution is on the destination host rather than the source host. The steps for migrating a process are as follows. The process is suspended immediately after the migration request. The bare minimum information needed to run the process is then transferred to the destination host. After this is done, the process resumes execution on the remote host immediately, even though it does not have all the necessary state information. In parallel with this execution, the remaining state data is transferred. If the resumed process needs some state information before it is available, it can make a request to the source host for that information. The reply to this request should have priority over the rest of the information being transferred, and it should be marked so that it does not have to be re-sent later. This requesting resembles Lazy Copy, but does not suffer the problem of residual dependency. It also resembles Pre-Copy, except that the execution on the source host is suspended immediately to reduce the load, and no information must be transferred twice. The delay between suspension and resumption is very short. The only drawback is its more complicated implementation and resulting higher overhead.

Flushing. Flushing is rather different from the rest of the process migration algorithms. It takes into consideration the possible availability of a third host, such as a file server. This file server can simplify the algorithm significantly, while reducing the time lag between suspension and resumption. After a process is suspended on the source host, all dirty pages are flushed to the file server, eliminating the need to transfer them. Other state information, however, must be saved and transferred to the destination host. The resumption can start immediately after this smaller amount of state information is completed. During the execution, pages can be retrieved from the file server using the standard fault handling mechanism. Performance of this algorithm depends on many system factors such as implementation of the file server, network speed, etc. The drawback of this method is that it only works in a homogeneous distributed environment where memory can be flushed into the file system.

4 Implementation

The implementation of process migration mechanism and policy involves a number of tradeoffs that need to be made with the requirements and constraints of the specific system in mind. The primary issues that must be dealt with in the implementation of process migration are location transparency, residual dependency, complexity, and performance [9]. As pointed out by Douglis and Ousterhout, these four factors are in conflict with each other and force design decisions to made to best achieve the goals for a particular system.

Transparency. Ideally, process migration should be invisible to a migrating process and to all processes that communicate with a migrating process. The Sprite operating system provided both of the above forms of transparency through four different techniques [9]. This was achieved by making kernel calls as location-independent as possible by using a common name space among the machines in the system, and by forwarding enough process state to the new host to allow normal kernel calls to be made after migration. For remaining kernel calls that could not be implemented transparently, those kernel calls were forwarded back to the home machine for the corresponding process. This implementation caused some residual dependency on the home machine, which resulted in a reduction in reliability by distributing the process state among multiple machines. However, transparency was considered to be more important in this implementation so this compromise was deemed acceptable.

Residual dependencies. When a process is migrated to a remote host, dependencies on the home machine may remain. These can be in the form of remaining process state that was not initially copied as part of the initial process migration but is available on demand, or kernel services that cannot be made transparent as described above. Residual dependencies add complexity to the system by distributing the state of a migrated process around the network, rather than concentrating the process state at the current host, which has the effect of reducing reliability by making processes more vulnerable to system failures. This also decreases the performance of the migrated process and adds to the load of the host machine by requiring remote operations rather than local ones.

Complexity. Since the process migration implementation involves many different parts of the operating system kernel, it is important to minimize the complexity of the implementation to ensure an acceptable level of reliability and maintainability. In the Sprite operating system, it was observed that, during development, the migration implementation continually broke due to other changes in the Sprite kernel, and that the maintenance load for the mature version was still higher than for many other parts of the kernel due to the relative complexity [9]. Friedman and Mosse also concluded from their experience with reliable, distributed servers that the simplicity of a process migration scheme was highly correlated with the chances that the scheme would actually be implemented [1].

Portability. In the implementations surveyed, the migration mechanism was typically separated from the migration policy. The migration mechanism is typically implemented in the kernel for efficiency, while the policy is implemented as a utility process for flexibility. As a result, the mechanism implementation is inherently not portable due to the dependencies on the specifics of the operating system kernel. The policy implementation is less coupled to the operating system kernel, and typically communicates with the migration mechanism through a set kernel calls specially added to support process migration.

5 Load metrics

A load metric provides the load distribution policy algorithm with a representation of a workstation's load. Since the load distribution policy depends upon this information in making decisions about which host to utilize or offload processes from, it is necessary for this metric to use relevant information. For instance, metrics based on committed events or Virtual Time are better suited for parallel discrete event simulations running on a Time Warp OS than a metric based on CPU utilization.

Some of the various load metrics utilized for general purpose operating systems are the following[12]:

- CPU queue length.
- Resource utilization.
- Communication delay.
- Host idleness.
- Object Affinity.

Resource Utilization. This approach bases the load on the utilization of one or more resources on a host. CPU utilization by itself is the most commonly used such metric. Gallard et al. [5] describe a multiple resource metric which is returned as a vector. The metrics used are:

- The current available processing power (CAPP) of a node which is a ratio between the processing speed (MIPS) and the number of processes currently serviced by the local CPU.
- The free memory size in the node.
- Disk transfers, which is a measurement of the data actually transferred on each disk and hence the internal I/O traffic.
- Network transfers, a measure of the number of packets on each network interface which indicates the external I/O traffic.

These metrics are based on research that indicates correlations between the metrics and the response times for processes.

Communication delay. The delay in a host's response to a load polling request is used as a measurement of its load in such an approach. The V migration system discussed in Theimer et al. [13] used such an approach. Hosts matching an availability criteria responded to a multicast poll request and the first response was selected. The assumption was that the first host to respond is generally the least loaded.

The advantage of this technique is that there is no additional overhead for broadcasting the load information separately and the network delay is factored in as well.

Host idleness. This metric is a simple Boolean value that is either idle or busy based on some idleness criteria. This approach is most useful in load-balancing algorithms for networks of workstations in which most hosts are largely idle.

Object Affinity. Reducing communication costs between processes can also a the goal for some load-balancing strategies. Communication can be treated as a measure of load, and minimization of this load requires communicating entities to reside together on the same host. Wilson and Nicol [14] use the number of messages sent between Logical Processes (LPs) in a parallel discrete event simulation (PDES) as a measure of communication weight which is factored into both the processor load calculations and into the assignment of LPs to processors.

Other metrics may be required for special cases such as PDES running on a Time Warp based OS where a metric such as CPU utilization is useless because it is possible that a processor has a high CPU utilization but that it is processing events ahead of the others and then rolling back. In such cases load metrics based on the number of 'committed events' are more useful. Deelman and Szymanski [15] present a load metric for such a system which uses a time weighted count of the number of events scheduled in the future of a process to base load calculations on. This approach differentiates between events that are scheduled to happen at different times by giving events scheduled with a smaller distance in time from the start of the simulation a higher weight than those that are further in the future. The advantage of this algorithm is that it bases its load estimate on the future of a *Logical Process* (LP) rather than on its past performance. Giving a higher weight to more immediate events reduces the load generated by rollbacks.

The load information generated is typically gathered by methods such as polling the hosts with requests to send back their load data, hosts broadcasting their load information to all other hosts or just to a group (multicasting) or a centralized collection policy under which a collection agent maintains load information for the network.

6 Conclusions

The research to date indicates that process migration is a viable methodology for improving load balancing and fault tolerance in distributed computer systems. The ability to migrate processes to be collocated with required resources allows system resources to be used more productively while reducing unnecessary network traffic. It is also clear that simplicity is a key ingredient in a successful process migration implementation, and that the migration policy and mechanism must be carefully matched to the specific system requirements and constraints to fully realize the potential benefits.

While process migration mechanisms are inherently non-portable due to kernel dependencies, process migration policies can be portable, particularly if a standard set of kernel calls that support process migration were to be available. Methodologies for heterogeneous process migration are also worth further research due to the potential flexibility to be gained if processes could be efficiently migrated between heterogeneous host machines. Since process migration policies select processes for migration based on predictions of process resource usage patterns, it would be worthwhile to look into the benefits of extending the process creation kernel calls to allow user processes to provide hints regarding the expected resource usage patterns for processes as they are created. This would allow programmers to assist the local process migration policy in making decisions regarding where and when to migrate processes for best results.

References

[1] R. Friedman and D. Mosse, "Load Balancing Schemes for High-Throughput Distributed Fault-Tolerant Servers", IEEE 16th Symposium on Reliable Distributed Systems, 1997.

[2] S. Petri and H. Langendorfer, "Load Balancing and Fault Tolerance in Workstation Clusters Migrating Groups of Communicating Processes", Operating Systems Review, 29(4), 1995.

[3] M. Richmond and M. Hitchens, "A New Process Migration Algorithm", Operating Systems Review, 31(1), 1997.

[4] H. Schrimpf, "Migration of Processes, Files, and Virtual Devices in the MDX Operating System", Operating Systems Review, 29(1), 1995.

[5] D. Arredondo, M. Errecalde, S. Flores, F. Piccoli, M. Printista and R. Gallard, "Load Distribution and Balancing Support in a Workstation-Based Distributed System", Operating Systems Review, 31(2), 1997.

[6] W. Zhu, P. Socko and B. Kiepuszewski, "Migration Impact on Load Balancing - An Experience on Amoeba", Operating Systems Review, 31(1), 1997.

[7] J. M. Smith, "A Survey of Process Migration Mechanisms", Operating Systems Review, 29(3), 1988.

[8] Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience", IEEE Computer, 22(9), 1989.

[9] F. Douglis and J. Ousterhout, "Transparent Process Migration Design Alternatives and the Sprite Implementation", Software - Practice and Experience, 21(8), 1991.

[10] A. Ferrari, S. Chapin, and A. Grimshaw, "Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Modification", Technical Report CS-97-05, 1997.

[11] V. Zandy, B. Miller, and M. Livny, "Process Hijacking", 1999.

[12] http://www.mcs.vuw.ac.nz/~kris/thesis.

[13] M. Theimer, K. Lantz, and D. Cheriton. Preemptable Remote Execution Facilities for the V System. In Proceedings of the Tenth ACM Symposium on Operating Systems Principles, 1985.

[14] L.Wilson and D.Nicol, “Automated load balancing in SPEEDES”, Proceedings of the 1995 conference on Winter simulation conference, 1995.

[15] E. Deelman and B. Szymanski, “Dynamic load balancing in parallel discrete event simulation for spatially explicit problems”, Proceedings of the 12th workshop on Parallel and distributed simulation , 1998.