

Issues in Process Migration

Sunil Thulasidasan
University of Southern California
thulasid@usc.edu
December 15, 2000

Abstract

*Process migration means moving a process in the middle of its execution from one processor or host to another, for a variety of reasons. Usually, processes are migrated with the aim of balancing the work load across the cluster so that the capacity of underutilized nodes are also exploited. The idea of process migration is borne out of the fact that in the over whelming cases of non migratory scenarios, the capacity of most nodes are under utilized. Allocation can be static or dynamic. By its very nature, process migration presents a difficult task with many complex issues to be resolved - transparency, scheduling and allocation policies, interaction with file system, naming, and scaling. In this paper we look at some well known implementations - **Condor**, **Sprite**, **MOSIX** and **MPVM**. We see that facilitating process migration adds considerable complexity to the kernel where it is implemented at the kernel level (Sprite, MOSIX) and hinders transparency if implemented at the user level. We also look at some analytical results regarding process migration benefits for load sharing. We see that the cost of migrating a performance comes mainly from the cost of transferring state. We conclude that migration, in the general case, should be used only sparingly, and often as a last resort.*

1. Introduction

Process migration refers to the act of disembodied an active process on the machine that it is currently running, and transferring its state to another machine, where it will resume execution from the point at which it was suspended. The concept of process migration was borne mainly out of two observations[1] - a network of autonomous work stations connected by a high speed network represents substantial computing power, much cheaper than a high performance single server, and that most of the time, the capacity of these work station is grossly under utilized. Thus process migration presents an intuitively appealing idea whereby a CPU intensive job can be moved from a machine, which may be already heavily loaded, to another one, which lies idle. If e after migrating a process, the process can resume execution oblivious to the fact that it has migrated, then migration is said to be transparent[2]. Transparency also means that program code does not have to be different for migratory and non migratory environments. Thus we see, that the fundamental goal of process migration is load balancing with transparency as an extremely desirable characteristic. In this paper, we shall look at the following implementations - Condor[3], Sprite[2], MOSIX[4], and MPVM[5]. We shall see how these systems accomplish (or fail to accomplish as the case may be) these goals, the trade offs they make, and the conclusions that can be drawn from the implementors' experience.

In the next section, we shall look at why process migration, at least theoretically, maybe a viable alternative to static load balancing. In section 3, we shall take a brief look at our case study systems

- their motivation, design goals and philosophy. In section 4, we shall go through, the various issues on which we analyze these systems - scheduling and allocation, interaction with file systems, naming, transparency and scale. In section 5 we shall look at the implementation of the migration schemes in these systems, and how they address these issues. Then in section 6 we shall take a look at some performance results of these systems when compared with non migratory or static schemes. In section 7, we briefly go through some analytical results in this field and how the results of the case studies match these. Finally in section 8 we conclude with our general observations on the suitability of process migration.

2. What can migration achieve?

It is possible that when a process is created, say, using the *exec* system call in UNIX, it can be statically allocated to a processor in the pool such that load is balanced. The problem arises due to the fact that loads can vary greatly over time, and generally it is not possible to have a priori knowledge of load distribution. Also, how would a target processor be chosen in the first place? Moreover, in a network of workstations, there is the concept of ownership of workstations [2], so it is appropriate that a process be evicted from a previously idle work station when the user reclaims it. Also, a static scheme wouldn't be transparent, implying that it would be up to the user or the process to decide which machine to execute on. Thus if the scheme has support from the kernel, then performance can be greatly improved[1].

The important features one hopes to achieve while implementing a process migration facility [1] would be automatic allocation and scheduling, with fairness, transparency, automatic eviction and overall performance improvement by load sharing. Of course, particular implementations have their own priorities with respect to these goals, for e.g., in the Sprite network operating system, transparency, eviction and performance improvement were the main considerations [6], while in MOSIX the prime consideration was dynamic load balancing[4], and in Condor[3] where the scheme was implemented at the user level, transparency was sacrificed for ease of implementation, load balancing and automatic eviction.

3. Overview of Case studies

The Condor system [3], developed at the University of Wisconsin, is a scheduling system for an environment of UNIX workstations, networked through a high speed LAN. The primary aim of condor is to identify idle workstations in the environment and schedule back ground jobs on them for load balancing. Also important was to evict the process as soon as the user reclaims the workstation so that the owner of the remote machine is not burdened by the additional load on his or her system. Condor is implemented entirely in the user space, by having programs linked to the Condor migration library routines during compilation.

The Sprite Network Operating System, developed at the University of California at Berkeley [7], represents an interesting case study of a system where migration facilities are part of the kernel. Transparency was a prime consideration here. Sprite was implemented in a fairly homogenous environment of Sun workstations connected over a very fast Ethernet, running the Sprite OS, whose kernel call interface was very similar to BSD UNIX [2], the important difference being a high performance sophisticated network file system and caching mechanisms. This, and other

aspects of the Sprite OS, as we shall see later, was very conducive to developing a migration facility at the kernel level.

The MOSIX system [4], developed at the Hebrew University in Israel, is a high performance cluster computing system consisting of networked work station of Pentium based PCs, connected by a high speed LAN. MOSIX consists of a set of additions to the UNIX system[8], with the primary feature being aggressive and dynamic load balancing. The pre-emptive migration scheme, implemented at the Kernel layer, has sophisticated mechanisms for "adaptive load sharing"[8], and transparent execution.

MPVM (Migratable PVM) [5] developed at the Oregon Graduate Institute of Science and Technology, is a migration transparent version of PVM. The PVM (Parallel Virtual Machine) [5] is a software system that allows a network of heterogeneous workstations to be viewed as a single virtual machine with distributed shared memory. In the original implementation of PVM, processes were allocated to processors statically. Migration allowed the usage of idle workstations in a dynamic manner. MPVM is implemented at the user layer. We shall also look at the performance of MPVM compared to PVM implemented over MOSIX.

4. Migration issues

Obviously, achieving the goals of load balancing and transparency with as low over head as possible presents a formidable task. The following are some of the main issues to be dealt with :

(1) **Allocation and scheduling:** How is a target node chosen? What are the factors taken into consideration while choosing a destination host? Is load balanced dynamically, or only reallocated during special circumstances like eviction or imminent host failure ? Does the previous history of allocation on a node make that node more attractive due to the presence of "warm caches" [6], also known as cache affinity scheduling ? Considering that all of the above systems represent loosely coupled environments, how much of a difference can such a consideration make? Similarly, what is the best allocation policy for an I/O intensive process?

(2) Once a target has been chosen, how is the **process state saved and transferred**? For e.g., would virtual memory pages be transferred all at once, increasing the latency between process suspension and resumption, or transferred on a demand-paged basis thus speeding up migration? An important consideration over here is how much of "residual dependency" [6] do we allow on the ex-host ?

(3) How is migration supported by the underlying **file system** for kernel level schemes? Are files assumed to be accessible from any point? For transparency, a transparent file system would itself seem to be a prerequisite.

(4) How are **name spaces** dealt with ? Do process Ids, file descriptors etc change with migration? How does global naming help? How are sockets and signals managed ?

(5) What are the **scaling** considerations that have been incorporated into the design ?

(6) What is the level of **transparency**?

We shall consider the case studies with respect to the above factors.

5. Comparison of Implementations

Allocation and scheduling:

In **Condor**[3], jobs are only allocated or migrated to remote work stations if the "leverage" [3] is significant, i.e. the ratio of remote capacity to the capacity needed at the local system to support remote execution should be large. Each work station has a local scheduler and a back ground queue, which holds the jobs submitted by the user. A central co-coordinator is present on one work station. Each station keeps information regarding its job load. The central coordinator uses a polling mechanism to see which stations are available, and allocates capacity. The work station decides the scheduling, according to the relative priority of the jobs. When the owner reclaims the machine the process is "check-pointed" (which we explain below) and state transferred to the home machine. The remote process does wait for an interval of up to 5 minutes, to see if the machine becomes available again, since reclamation periods tend to be short. The main points to be noted here are that Condor does not really take into account affinity based scheduling. Whether such a policy would make sense in a loosely coupled system where all remote resources are freed up and no residual dependencies remain is debatable.

In **Sprite**[6], processes are migrated during two occasions - when a resource intensive program is about to start, or during eviction from a remote host. Host allocation is done using a combination of a load-monitoring daemon on each host and a centralized idle-hosts database file. Idle hosts are selected based on the history of idle time length. This factor is given more priority than the presence of "warm caches" since eviction represents a greater cost. Thus host or cache affinity is not considered in Sprite. After eviction, the process is allocated to another host if available. Also, Sprite has the policy of leaving "minimal residual dependencies" on the ex-host [6], meaning all state is transferred out as soon as possible.

MOSIX [10] has the most complicated scheduling mechanism among all the systems. This is because MOSIX differs in its design goal in a fundamental way - load balancing is done continuously, not just during creation or eviction of a process. Processes get migrated anytime the cluster gets unbalanced, through adaptive scheduling[4]. If a process requirement exceeds a certain threshold, then a process becomes a candidate for migration. Each process must also run for a bare minimum time on the processor to prevent thrashing. A load vector is maintained at each node, which contains information about the load of a random subset of neighboring nodes. This load vector is constantly updated through "load information dissemination" [4] which is a completely decentralized process. Candidate target nodes are chosen from this load vector. During allocation, I/O bound processes are allocated on nodes with which this process has maximum I/O communication (I/O affinity). Also, a process that has a history of forking other processes becomes a good candidate for migration.

In **MPVM**[5], a unit of work is known as a 'task'. MPVM uses a Global Scheduler (GS), which is basically a centralized resource manager. The GS decides the 'which' (task), 'when' and 'where' (target node) of migration. Migration is usually done during work creation and eviction or when a node is under excessively heavy load (Dynamic allocation). Idle hosts are located by the GS in a manner somewhat similar to Condor. Each node has a PVM demon (pvmd) installed on it. When a decision has been made by the GS, a signal is sent to the pvmd on the node from which the process

has to migrate. Target allocation is based on idle work station availability. MPVM does not use cache or I/O affinity considerations.

Transferring of State:

Migration in Condor is basically a combination of the "Remote Unix" (RU) facility combined with "check-pointing" capability [3]. When RU is invoked, a "shadow" process runs as a "surrogate process" on behalf of the remote process on the home machine. System and other location dependent calls are forwarded to the surrogate process. The "check-pointing" facility is to save the state of the process, so that the process can be restarted elsewhere. Saving the state involves writing the process's data and stack segments to permanent storage using the file system [9]. In a uniform file system environment, this is equivalent to migration (since the file system is mountable from any host), while in environments where the nodes do not have a uniform view of the file system, calls are forwarded via RPC to the shadow process on the home node, and the results are sent back[9]. Since condor is implemented at the user space, to access process state data, a "check-pointing" library is used to give ability to a process to check point itself. Condor cannot save the state of IPC structures like sockets, pipes and signals.

In Sprite [6], since migration is implemented at the kernel level, considerable amount of effort has gone into supporting migration. Most of the migration overhead involves the transfer of virtual memory. Remember that Sprite uses a network wide file system. Virtual memory is frequently written to backing storage. Also, during transfer, dirty pages are flushed out of the cache. The virtual memory file is then remounted at the new host on a demand-paged basis.. Other process state info like openfile descriptors, file handlers , message channels etc represent less overhead.

In MOSIX [4], once a target node has been picked there is an exchange of messages between source and destination. The destination node can choose to reject the request. MOSIX uses a demand paged transfer of virtual memory. Because of kernel level implementation, it is easy to store the process and processor states. Even though the environment is generally hardware-heterogeneous, migration is allowed only between homogenous processors. In fact, process migration is the only feature in MOSIX that is restricted by homogeneity.

In MPVM [5], which is a user level implementation, a process establishes a TCP connection with the destination node. All process state that can be captured by the application using the pvm library functions is transferred to the destination and a skeletal process is constructed. Note that even though PVM supports heterogeneous clusters, migration can only be performed between homogenous nodes in a cluster. OS specific state cannot be migrated due to the fact that much of it is not observable by the user level implementation.

File System and Migration

As mentioned before Condor[9] can support both uniform and non uniform views of the file system. In an NFS like environment, where any file can be remotely mounted, check point/restart is simpler to implement. File state information such as open file descriptors, seek position etc are captured at check point time. It also supports non uniform view of the file system through forwarding mechanisms via RPC when Condor is implemented over a wide area network. One important assumption is that the state of the checkpoint file is not altered between checkpoint and restart.

Sprite uses a globally named uniform file system, thus file access is completely transparent across nodes. The cache flushing mechanism of the file system is used for transferring virtual memory, as well as for keeping a consistent view of the file system. As we shall see, the most important aspect of transparency in sprite comes from its global naming scheme.

MOSIX uses the UNIX file system, and thus, has a uniform transparent view of the file system. This facilitates transferring of virtual memory files.

MPVM assumes that a global file system like NFS exists on both source and target nodes. This is required for file I/O migration to work in MPVM. To accommodate file I/O migration, a set of wrapper functions are provided in the PVM file I/O library. This allows the PVM library to maintain a list of open file descriptors. However, note that, unlike Sprite, the file descriptors will change across machines.

Naming

As seen above, file name spaces in Condor can be global or non uniform. Also due to lack of global names for objects such as sockets, pipes etc the state information regarding these objects cannot be migrated.

Sprite uses a completely global naming scheme - Files, devices, process ids and even communication schemes are globally named. Files are logically centralized but physically distributed [6]. An interesting aspect in Sprite is that IPC mechanisms are also completely transparent because of this naming scheme. Objects communicate with each other using a "*pseudo device*" [6] which is an additional layer of abstraction thrown in to support transparency. Only the kernel is aware of the actual location of the pseudo device. Thus for e.g., even migration of socket IPC which is typically unsupported in other schemes, poses no problem to Sprite.

MOSIX too uses a global naming scheme which makes the implementation transparent. Universal-to-local mapping routines are implemented for managing file names.

MPVM assume a global naming scheme for files, even though PVM itself may run across a heterogeneous cluster. File descriptors, process Ids and signals are not global and not migrateable, thus hindering the goal of transparency in MPVM.

Scaling Considerations

We see that in Condor, there is a centralized coordinator who does the allocation[3]. However each node is also autonomous since it only needs to keep track of its own load state. If the coordinator fails, new requests are affected, not the requests that are already allocated. These aspects give Condor a certain degree of scalability

Sprite can scale up to a few hundred work stations [2] Further scaling is limited by the fact that a centralized idle-hosts database file is used.

MOSIX scales well due to a variety of reasons [4]. First, nodes are completely autonomous, and the scheduling is totally decentralized. Each node maintains information only about a random

subset of nodes, usually those at close physical proximity due to I/O affinity considerations. Each processor also sends out information regarding its load to only a random sub set of processors. All communication is carried out only between the concerned 2 nodes during migration. MOSIX also has what is known as "*Architectural Symmetry*" [4] whereby services are distributed across nodes. Moreover each node need have only partial information about the whole cluster.

MPVM uses 2 party communication for transferring state, however scaling is limited by the fact that a centralized resource manager (GS) is used.

Transparency

Condor is not really transparent to the user, since applications have to be linked with the Condor library routines. However the actual check pointing and restarting procedures are transparent to the process But the lack of a truly global naming and user level implementation hinder transparency.

Sprite, in contrast, represents a truly transparent scheme due to its global naming scheme of files, devices and IPC mechanisms using pseudo devices. For e.g., file system transparency was maintained by keeping the object that manages the name of the file (file server) separate from the object that contains the file (I/O server)[6] Transparency was one of the most important goals Sprite, even at the cost of additional complexity.

MOSIX is completely transparent, due to its naming schemes. Location dependent calls are forwarded to the home nodes. However, MOSIX also has the ability whereby the user process can explicitly request to be migrated using the *migrate()* system call, which also takes the destination node as its argument. When this happens, the automatic scheduling scheme is bypassed. This feature is useful for benchmarking the performance for MOSIX.

Even though MPVM strives to achieve transparency, it is not really implemented due to the fact that process state information like process ids that are known and used by the application process change upon migration.

6. Performance Results

Condor performance results indicate that it is ideally suited to long running computationally intensive jobs [4]. Long jobs are check pointed less often, since they finally end up on a station experiencing no activity. Condor is not suited to short jobs.

In Sprite, since evictions are usually infrequent, most of the latency observed by users is due to the overhead associated with remote execution. This is similar to active migration, except that no virtual memory is transferred. The best results for Sprite were achieved when using the pmake program when, using 10 hosts was about 5.5 times as fast as when using a single host. It is worth noting that migration has been one of the most fragile aspects of Sprite [1], leading to frequent kernel breakdowns. This often happened because different kernel versions were used. As a result, migration was later disallowed between kernels of differing version numbers.

A study of the performance results of MOSIX [4]shows that performance lags the static optimal balancing of loads by only a factor of 2%.

MPVM does not use check point migration [5]. The entire virtual address space is transferred on migration in contrast to a demand-paged scheme, which adds to the latency till restart. Performance results have shown that this is the dominant cost in migration. A comparison of MPVM with PVM over MOSIX shows that the native support for migration in MOSIX results in a much better performance than MPVM.

	Condor	Sprite	Mosix	MPVM
Scheduling and allocation	Centralized allocation. Individual scheduling. Migration only during creation or eviction	Uses centralized file for host search. Migration during process creation and eviction	Completely distributed allocation system. Highly dynamic load balancing	Centralized allocation. Individual nodes negotiate. Migration during creation, eviction well as dynamic
File System	Uniform file system – state transferred. Also supports non uniform file system through RPC	Network wide transparent , uniform file system similar to NFS	Network wide file system similar to NFS	Assumes existence of NFS (or other global file system)
Naming	Global name for file systems. Localized names for process ids , file descriptors etc.	Global naming scheme	Global naming scheme	File names are global. Process ids, descriptors etc are not
Transparency	Not fully transparent	Fully transparent	Fully transparent	Not fully transparent
Scale	Scalable, but uses centralized coordinator	Scalable to few hundred work stations, since central host-file is used	Very scalable. Has completely distributed policies	Scalable to certain extent. Central resource manager vs autonomous node communication.

Table 1: A summary of comparisons.

7. Analytical results in Process Migration

Eager et al [10] have shown analytically that migrating active processes for the purpose of load sharing offers only modest performance benefits over a non migratory load sharing scheme, and that too under fairly extreme conditions. Moreover, they also show that migratory load sharing never yields major performance benefits. This is generally consistent with the performance results of the systems we have considered, whereby even in the best case, performance benefits of only 25% were achieved. Efficient migration is indeed a tough call, since by its very nature migration presents considerable overhead. However, load sharing may not always be the only aim for migration. Evicting processes so that the machine resources may be reclaimed by the owner is an important motivation. Also, there can be cases where a process has to be migrated from a host due to imminent host shutdown. From a point of view of load sharing too, it is not hard to see that long running computationally intensive jobs with infrequent migrations, stands to benefit from a migratory load sharing scheme.

8. Conclusions

It is worth quoting Fred Dougkis, [1] who implemented the migration facility in Sprite, at this point.

"..Sometime ago shortly before the file system was to be reimplemented, we had a lengthy discussion about the future of process migration in Sprite. The consensus at that time was that migration was probably a mistake : it was too difficult to implement. ...In retrospect, I may safely say that our initial lack of faith was misplaced. Process migration has evolved from a toy prototype to a mature, extremely useful facility. Users are thankful not only for the significant improvement they see when using their hosts, but also for the minimal impact that other users have on their work stations."

Process migration is difficult. Providing efficiency and transparency is even more difficult. A global naming scheme is seen as an essential ingredient for providing transparency [1], and hiding remote execution. Techniques like affinity scheduling are out weighed by other considerations like time to eviction in a loosely coupled environment where migration is usually implemented. There are scenarios, like long running jobs or extremely variable dynamic loads, where the pay offs may justify the efforts. However these do not represent the general case. If an optimal or nearly optimal static scheduling can be done apriori, then a migratory load sharing scheme is not viable. The consensus at this time about migration is that, generally, it should be used only as a last resort.

9 Acknowledgements

I would like to thank Dr. Neuman and Dr. Obraczka for their timely feedback on my proposal on this topic.

References:

1. Fred Dougkis: Experience with Process Migration in Sprite. In Distributed and Multiprocessor Systems Workshop Proceedings, pages 59--72, Fort Lauderdale, FL, October 1989. Available on sprite.berkeley.edu or here on ftp.ibr.cs.tu-bs.de
2. F. Dougkis and J. Ousterhout: Transparent Process Migration: Design Alternatives and the Sprite Implementation. In Software -- Practice and Experience, volume 21, number 8, pages 757--785, August 1991. Available on sprite.berkeley.edu or here on ftp.ibr.cs.tu-bs.de
3. M Litzkow, Miron Livny, Matt Mutka Condor - A Hunter of Idle Work Stations, Proceedings of the 8th. Int'l Conf. on Distributed Computing Systems, 104-111, 1988.
4. Amnon Barak and Shai Guday and Richard G. Wheeler: The MOSIX Distributed Operating System. LNCS 672, Springer, Berlin, 1993.
5. J Casas, D Clark, R Konoru, S Otto, R Prouty, J Walpole : MPVM: A Migration Transparent Version of PVM, OGI Technical Report, Feb 1995
6. Frederick Douglas : Transparent Process Migration in the Sprite Operating System (PhD Thesis, University of California, Berkeley), September 1990
7. J Ousterhout et al : The Sprite Network Operating System (IEEE Computer, February 1988)
8. Annon Barak and Oren La'adan: The MOSIX Multicomputer Operating System for High Performance Cluster Computing, Publication of the Institute of Computer Science, Hebrew University of Jerusalem

9. Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny: Checkpoint and Migration of UNIX Process in the Condor Distributed Processing System, Computer Science Department, University of Wisconsin, Madison
10. Amir Y., Awerbuch B., Barak A., Borgstrom R.S. and Keren A., An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster (ftp), IEEE Tran. Parallel and Distributed Systems, Vol. 11, No. 7, pp. 760-768, July 2000.
11. D. Eager and E. Lazowska and J. Zahorjan: The Limited Performance Benefits of Migrating Active Processes for Load Sharing. In Conf. on Measurement & Modeling of Comp. Syst., (ACM SIGMETRICS), May 1988, pages 63--72.