CS 655: Advanced Topics in Distributed Systems
Paper Critique 3
Hessah Alsaaran

Paper


Summary

Process Migration can be defined as “the act of transferring a process between two machines during its execution”. Processes can be migrated to either (i) dynamically balance the load between the nodes of the system, (ii) move it to the node that owns a local copy of the data, (iii) move it to the node that it frequently communicates with, (iv) recover from failure when checkpointing is used, or (v) evacuate a node for administrative shutdown. This paper is a survey about process migration, describing its concepts, along with four systems as case studies: MOSIX, Sprite, Mach, and LSF.

The general main steps for process migration consist of suspending the process when its migration is accepted by the destination node, transferring the process’s state, and then resuming its execution. Although there are variations in these steps in different algorithms, there are two main costs that should be minimized. The first is the process’s suspension time, while the second cost is the amount of data to transfer. These costs have been addressed by the different transfer strategies discussed in the survey. The information sent at migration time constitutes the initial migration cost, while the transfer of the remaining information constitutes the run-time cost. However, there is a trade-off between the initial costs and run-time costs, and a correlation between the initial cost and suspension time. In addition, keeping some information on the source node introduces a dependency problem, compromising failure-tolerance of the system.

Since the migration process is costly, it should be justified in order for it to be performed. One of the main goals of process migration is dynamic load balancing. To support migration decisions, load information should be maintained. Different systems vary in the information used. However, there is a trade-off between the size of this information and the incurred network overhead for disseminating it. In addition, there is a trade-off between the frequency of releasing this information per node and the stability of this information. Another variation is how systems manage this information and perform the migration decisions. Some systems use distributed algorithms, while others use centralized control. Distributed algorithms could be more scalable, but centralized systems could support better migration decisions.

Critique

From the definition of process migration, stated above, the reader may assume that when the process is transferred, it implicitly resumes execution. At least this is what I thought at first. But when the survey discussed alternatives to the process migration approach, two alternatives do not support process resumption, while one of them is actually a mechanism for performing process migration. So either these two alternatives are not really alternatives (or special case alternatives: for very short processes that are not worth transferring their state), or the definition needs more clarification.

The first discussed system is MOSIX. MOSIX strives for transparency, and that affected many design issues that I don’t agree with. For example, when a process is migrated, it is not fully migrated as there is a shadow for it in its source node that services the process during its execution. So there is always a dependency between the source and the destination nodes, which can compromise failure-tolerance of the system. They have chose to do so to hide the migration from the process so it continues to execute as if it didn’t migrate. And I guess also to use the eager (dirty) transfer strategy for reducing initial migration costs and reduce process suspension time. However, the case of a source or destination node failure is not
discussed. Although transparency is an important quality for any distributed system, in my opinion, the ability to cope with failure is far more important. In addition, this design choice also incurs management and network overhead, because of the dependency between the source and the destination nodes, limiting the applicability of this system to compute-intensive processes. It is also not suitable for administrative shutdowns of nodes, since processes are not fully migrated.

A good feature of MOSIX is its distributed load management and migration decisions, as it gives MOSIX great scalability. However, I don’t agree that representing a node’s load with the number of load units is enough. That is because not all loads are equal. Many short processes could be equivalent to a long process. In addition, a node should have a mixture of CPU-bound processes and I/O processes to effectively utilize its resources. As a suggested possible solution, loads can be weighted by their estimated running time. However, this estimation computation could be an issue. One way around it is using the history of executions in the case that it has been recently run on the node. Regarding the mixture of processes type, the system can try to maintain a configurable ratio of CPU-bound processes and I/O processes. This information about processes can be extracted from their profiles.

In MOSIX load information dissemination, every node divides its load information in half and sends each to two randomly chosen nodes. I think that the number of nodes to send information to can be adaptive in a way that it increases as the cluster grows in size. This can promote faster dissemination when the cluster size increases.

A limitation of MOSIX is that it can only be used with private Linux clusters that contain only trusted and authorized nodes linked through fast network. This choice may have been made to avoid heterogeneity and security issues. However, it limits the applicability of the system.

Sprite, similarly to MOSIX, also achieves transparency through the use of home-nodes. So it inherits the same problems associated with this decision. Unlike MOSIX, its goal is utilizing idle nodes in the system instead of full load balancing, although the latter case covers the first one. In addition, it used a centralized load information management and migration decision. This can cause a bottleneck as the system scales up.

Mach has an added task migration feature. It is similar to the previous two systems in that there is a dependency between the source and destination after migration. So it also inherits the same problems associated with this decision. However, its scheduling is distributed. Since it uses a distributed file system, it considers data locality in addition to load balancing for migration decisions.

None of the previous three systems considered initial placements of processes. They were always tied to their source nodes. In LSF (Load Sharing Facility), the primary goal was the initial placement of the processes, considering load balancing. Then it copes with dynamic load changes using process migration combined with checkpointing. The disadvantage in this system is the centralized control, where a single master is responsible for scheduling and maintaining the system’s state. This is a single point of failure. However, the authors said that when this case happens, another node would promote itself to be master, although I’m not sure how the selection is performed.

To summarize, there are full vs. partial process migration systems. MOSIX, Sprite, and Mach provided partial migration, while LSF provided full migration. In partial migration, the dependency between the source and destination nodes increased transparency, but at the same time incurred management and network overhead. Also, it compromises failure-tolerance of the system. These issues can be alleviated using full migration. It can cause longer migration time, but less run-time overhead. However, process and node compatibility becomes an issue here. So more information needs to be disseminated in order find matches. This in turn incurs more network and storage overhead.

Also, to complete the summarization, there are distributed vs. centralized load balancing systems. MOSIX and Mach are distributed, while Sprite, and LSF are centralized. Distributed algorithms are more scalable, but on the other hand, centralized algorithms get fast and up-to-date information about all nodes
supporting better migration decisions. However, the later can be as single point of failure or a bottleneck when the system scales up. A possible issue for distributed algorithms is the storage of load information; different nodes have different storage capabilities.

Finally, one question always asked in class is: if I would build a new system, how would I do it? Well, there is a lot of trade-offs involved, and selecting which side to choose is really application/domain dependent. It also depends on the resources available to you. But if I were going to build a general system, then I might choose the following.

First, I would like to do load balancing through full process migration (no home-node dependency) to achieve more failure tolerance. In addition, I would do checkpoints at application level. This will harm transparency, but it produces significantly smaller checkpoint files that are also machine-independent. Then replicate these checkpoints at a configurable level on different nodes. These checkpoints will be periodically saved and pushed to update replicas. If a node holding a replica does not respond, a replacement is chosen. On the other hand, if an update timeout occurred at these nodes, one will take over the process. When a process has terminated, the node informs the nodes holding replicas to delete these replicas.

The second choice is to do migration decisions in a distributed manner as in MOSIX since it is a scalable approach. But I would change the load representation to instead of basing on only the number of units as if they are equal: Load units will be weighted by their expected running time, in addition to keeping a configurable ratio of CPU-bound and I/O-bound processes. Also, instead of sending load information to two random nodes, let the number of random nodes increase proportionally with the scale of the system. These issues and possible solutions were previously discussed above.

The third and final choice is to implement the system in the application-level as in LSF. This choice increases portability and simplicity as apposed to OS-level implementation. Portability will increase the system’s applicability, while simplicity motivates further advancements and improvements. On the other hand, the system’s performance may get degraded.