Transparent Process Migration...

Fred Douglis
John Ousterhout

Michael Bowling

Goals

Process migration in sprite was developed to make use of the idle machines on a network of workstations. They wanted to exploit this opportunity without violating the notion that a user owns his workstation, and should have access to its full resources when in use. This was complicated by the fact that Sprite is Unix-like operating system that makes use of kernel calls.

There are four goals of process migration: transparency, few residual dependencies, performance, and simplicity. Transparency is to allow both the process and user to behave normally after migration occurs. Residual dependencies are parts of a process' state that remains on its original machine. This causes both performance problems, and opens up a number of points of failure for the migrated process. Performance is of course the goal in doing process migration. Simplicity in the implementation allows it to be maintained more easily, which is important since process migration effects and is affects by many aspects of the system. The Sprite system focuses on transparency and reducing residual dependencies.

Managing State

The state that is associated with a process includes:

- Virtual memory
- Open files
- Message channels
- Execution state
- Other kernel state

A process migration system must handle how the pieces of the process' state is available on the target machine. There are three ways of managing a piece of the process' state. It could be transferred to the target machine, which is the obvious choice for the execution state. Requests for the state could be forwarded to the source machine, which is necessary for state using a specific piece of hardware. The last option is to simply sacrifice transparency. This last option makes sense for some kernel calls (e.g. requesting the amount of available physical memory.)

In Sprite, message channels are implemented by sharing an open file in the network file system, and so do not need to be managed specifically. Much of the kernel state is simply transferred, although enough information is maintained on the source machine to locate the migrated process. This leaves virtual memory and open files as the difficult pieces of state to manage.

Virtual Memory

A number of techniques to handle virtual memory are compared. The various techniques tradeoff performance and residual dependencies. Monolithic transfer at migration time has no residual dependencies but takes a huge performance hit since the process is frozen while the memory is transfered. Pre-copying can reduce the freeze-time, but will have to copy in-use pages twice. Lazy-copying only copies pages once on-demand, but leaves residual dependencies on any machine the process has migrated through.

Sprite's technique makes use of their network file system, since the virtual memory backing store is maintained in the file system. The source machine simply flushes its dirty pages, and the process at the target machine will load in the pages from the file system as demanded. This is similar to lazy-copying, but no residual dependencies are left on the source machine, since they are stored by the file server.

Open Files

This was one of the unexpectedly difficult parts of the process' state to manage. Many of the difficulties were in maintaining the semantics of the file system. The difficult things to maintain were the Unix semantics that an open file would not be deleted until it was closed, and processes (child and parent) could share access positions of open files.

An open file has three pieces of state associated with it: the file reference, caching information, and access position. All three pieces of the state brought up difficulties. The file reference could not be closed and opened again on the target machine, because the file system could delete the file during the migration. Their cache rules disable all local caches if a file has two writers. During migration of a file open for writing, both the target and source machine will be writers of the file. This unnecessarily disables caching. Lastly, since a child process could migrate to a new machine, how can they share the access position.

All of these difficulties were solved by modifications to the file server itself. They implemented a protocol where the source machine transfers the open file information to the target machine, the target machine then informs the server of the transferred files, and finally the server informs the source machine that the transfer is complete.

Migration Policy

The other major question to be answered is how migration should be used. What programs should migrate? When should they migrate? Where should they migrate? Who decides these answers? Answers to these questions vary from the user deciding everything (e.g. `rsh`) or everything's automatic (e.g. "pool of processors" model.) Sprite takes an intermediate approach.

Applications and users are required to determine the programs that migrate. For example, `pmake`, decides which portions of the compilation can be made in parallel and migrates processes to perform these steps. The system decides automatically where processes migrate. This is done by maintaining a server that keeps track of idle machines, and a daemon on each machine is responsible for informing the server when it is ready to take processes. Lastly, when processes migrate is either at `exec`-time or due to eviction (when a machine no longer becomes idle.)

Performance

The Sprite system with process migration was used regularly for a two year period. Unlike other systems it saw extensive practical use. Performance was analyzed at both the application and system level. Below is a summary of the key performance results:

- Migration overhead is around 300ms, which is probably too large to migrate trivial processes, but warrants migrating longer CPU-bound tasks.
- Most non-contrived applications, especially CPU-intensive ones (e.g. compilers and text-formatters), had a miniscule performance-hit when running remotely.
- Using `pmake` improved compiling time by a factor of 3-5.
- 31% of all processing was done remotely, during a typical month.

Their final conclusion is that the performance increase and transparency made the system effective and got a great deal of practical use. The process migration system still required a great deal of kernel maintenance, as the system relied upon and effect many aspects of the kernel.