

General-purpose Process Migration

(Really Migrating Real Processes)

1. Introduction

This paper takes a new look at an old topic—process migration. Process migration has been heavily studied, prototyped and implemented—often with no real practical benefits. Traditionally, process migration mechanisms have been used to load balance processors in a distributed system and approaches for supporting them transparent to the application has required extensive kernel support. However, despite some isolated situations, preemptive load balancing has been found quite unnecessary in most computing applications. Consequently, the conventional view has held that the costs of supporting process migration mechanisms, in terms of both run-time overheads and operating system maintainability, are not justified by their potential benefits.

We feel that the implied criticism of process migration research is more a statement about the nature of these earlier approaches and target application scenarios rather than any fundamental shortcomings. We have found that the ability to migrate normal (shrink-wrapped) applications, servers and GUI based programs can have surprising benefits. Process migration can be a very effective tool for emerging applications, enabling capabilities such as mobility, collaborative work, distributed systems management, automatic reconfiguration, and fault-tolerance [MSD99]. Moreover, implementing general-purpose process migration need not require any kernel modifications to the underlying operating system. Although, migrating a shrink-wrapped application relying only on user-level mechanisms, without access to its source code, does not appear feasible, it is indeed possible given the application programming interface (API) interception technology that is currently getting widespread attention. Our approach to process migration involves a new scheme called *virtualization* that effectively decouples an application with dependencies to the operating system and the physical environment. Virtualization also monitors and controls the interactions and the resulting side effects between processes and operating systems, supporting process migration by facilitating transfer or recreation of necessary state. We have found that de-coupling the application from the base operating system and also de-coupling it with connections to peripherals (file handles, network sockets, graphics handles) can give rise to new “virtualized” environments where processes can freely migrate and reassign logical devices on the fly [B*99B, DKK99].

In this paper we describe the construction of a prototype process migration system using API interception mechanisms, outline recent experiences with migrating Win32 processes running on Windows NT, and identify several research issues that require further exploration. The thrust of this proposed fundamental research is to first form a model of the communication between an operating system and its applications, and then to experimentally validate this model, leading initially to a virtual process execution environment and finally to the ability to migrate a general process anywhere.

2. Process Migration – The Traditional Perspective

Techniques:

Traditionally, process migration has used one of the following two approaches:

1. The process is linked to a user-level migration library, which handles process state maintenance and migration facilities. This method typically handles migration only for “stand-alone” processes, restricting what the process can actually do while it is executing. For example, the process cannot be involved in GUI interactions, open network connections, or accessing file systems. These processes are migrated by saving the state of the process (or checkpointing the process) at some well-defined points and then later restoring this saved state. As an example, the Condor system [LLM88] only migrates compute-intensive processes without any GUI interactions or open network connections assuming support for remote file access using a network file system.

This method is also applicable to worker processes in a parallel processing system, which also rely upon a similar checkpoint/restart strategy. Example systems include the Emerald [JHB88], Charlotte [B*99A], and Chime [SD98, SMD98] programming systems, as well as migration-capable extensions of PVM [Sun90] such as MPVM [C*95] and DynamicPVM [D*94].

2. The inner core of a process and any kernel state related to this process is migrated using operating system mechanisms for migration, optionally leaving behind a proxy on the source site to handle any “residual dependencies” (e.g., active network connections). Such a strategy can migrate any general process, but incurs significant run-time overheads in addition to operating system complexity and maintainability costs. Moreover, completely eliminating residual dependencies is not possible unless the underlying operating system provides a network-wide identifier space (for process IDs, communication handles, etc.) and a network file system. Most distributed operating systems with support for process migration such as Chorus [R*92], MOSIX [B*93], Amoeba [T*90], SPIN [Be*95], and Sprite [DO91] fall into this category.

More efficient implementations of this strategy are possible in operating systems such as Solaris-MC [K*96] running on symmetric multiprocessors (SMP) or cache-coherent shared-memory machines because of additional hardware support.

Benefits:

Despite the large number of process migration prototypes described in literature, relatively few applications have been shown to benefit from these mechanisms. The first strategy described above has been more popular, with many programming systems and batch environments relying on it for spreading computations across a network of machines. The second general-purpose strategy has seen less use. Several of the distributed systems referred to above have used process migration for load-balancing jobs in a network, and parallelizing coarse-grained applications such as pMake (parallel compilation) and certain scientific computations, with results in literature reporting good throughput improvements. Unfortunately, these applications are no longer viewed as compelling, particularly given the ready availability of small- to moderate-sized SMPs.

3. Benefits of Process Migration—A Modern Perspective

Given current-day system configurations and emerging applications, general-purpose process migration affords significant far-reaching benefits, making it useful for much more than just load balancing:

- Consider a user actively using a complex application (such as a spreadsheet with lots of macros) that now wants to be able to work on another computer at another location without closing the application, needing therefore to migrate the application. In another scenario, the user leaves the

application running at work and then goes home and realizes she needs to do additional work. Process migration support would enable her to simply move the application over to the home machine. *The seamless migratability of applications between desktops and laptops can add another dimension to mobile computing.*

- If we can migrate the application and its screen and its active connections to networks and files, then using the same mechanism, we should be able to move the screen without moving the application process. Decoupling the various external interfaces of a process (GUI, network connections, files, etc.) from the internal state of the process facilitates many interesting collaborative work situations. Taking this one step further, decoupling the internal process state from its interactions with operating system components such as dynamically-linked libraries (DLLs) permits on-the-fly replacement of DLLs.
- The essence of process migration is the ability to capture all of the state that describes process behavior. Given such capabilities, application functionality can be extended using novel abstractions such as “active versioning”. For example, while a user is working on developing complex macros for a spreadsheet, he might decide to do some risky experimentation without necessarily saving the current state (not just the files but the entire environment). It is only later that the user decides whether to commit these changes or revert back to an earlier state.
- Consider a server running a set of objects being actively used by a lot of external clients over the Internet. The system administrator needs to shut this machine off for maintenance but does not want to disrupt the service. The server along with its state and even active network connections can be migrated to another machine.

This focus on the novel capabilities provided by process migration mechanisms motivates a different approach. Our goal is to be able to cleanly migrate any application process (without leaving any trace behind), irrespective of the behavior of the process. We do not want access to source code of the application, to modify the application or to re-link the application. While achieving this *may* seem like a lofty goal, we next describe our proposed approach that meets these requirements.

4. User-level General Process Migration

4.1. Solution Overview

The execution of a process is accompanied by two kinds of effects—the effects of the process on the environment and the effects of the environment on the process. The fundamental challenge of general-purpose process migration is ensuring that, from the perspective of the process, these effects appear consistent on both the source and target machine. A process that neither produces side effects nor accepts any input from the environment is quite simple to migrate. However, in general, a process relies on proper handling of effects such as:

- *State in the operating system and underlying hardware:* The opening of a file causes the operating system to create an entry in a resource table and return a handle. This side effect generates state in the operating system. Similarly, the program output changes the contents of the frame buffer and the state of the screen.
- *Interactions at the process-OS boundary:* Messages sent on the network causes the state of buffers in the operating system to change. Similarly receiving network traffic has side effects. Along the

same lines, a process accepts effects from the environment. For example, it receives messages from the GUI system and knows to repaint, to shrink or to hide.

- *Internalization of the environment*: A process gets the IP address of the machine it is running on. Now it has bound itself to that machine and will not be able to run on any other IP address. Similarly, a process might internalize its process ID, environment variables, and registry settings.

Program side effects are necessary for its proper operation. Our overall approach for ensuring that these effects are identical on the source and target nodes, to the extent detectable by the process, relies on a three-step procedure: First, figure out all the side effect the process has had on the operating system and recreate these effects after the process has moved to a different operating system. Second, ensure that any internalized state within the process continues to be valid after move. Third, ensure that after the move, further side effects continue to have the effect desired by the process.

The implementation of these steps depends on whether the interactions in question are to *location-independent* components of the system or to *location-dependent* components. Location-independent effects include file access in networked file systems, and network access between two controllable processes. For example, if a process is reading a file from location A, it can be moved to location B and made to read the same file from the same file position if the file is also accessible from B. Similarly if a process running on A has a network connection to B, and subsequently the process is moved to C, then the connection can be recreated as a connection from C to B. *Location independent side effects do not need the use of proxies for migration.*

Location-dependent effects on the other hand require proxies for proper handling. An example of such an effect is screen output. Consider a process initially running on machine A that is moved to machine B with the requirement that its output remain on A. This requires that the output subsequently generated by the process be routed to A, and a proxy on A display the output. In addition, all process IDs, IP addresses, handles and such have to be “faked” or rather virtualized such that they indeed do not change as the process moves. Thus, we use global values for all location-dependent input that the process receives from the operating environment.

The management of all of the process effects is achieved using a unified concept called Virtualization (Section 4.3). Virtualization creates logical connections from the process to external devices (files, networks etc.) Thus, we decouple the side effects from the physical resources, monitoring these interactions such that the side effects can be recreated as necessary. The mechanism used to implement virtualization is API interception (Section 4.2). API interception permits the association of arbitrary activity for every communication between the process and the operating system, thereby enabling the virtualization system to properly acquire and modify the side effects, and recreate their effects.

4.2. Background—API Interception

API interception is the ability to sever the bond between the application and the underlying runtime system (Win32 Subsystem in the case of Windows NT) and divert calls made the system API's to a set of routines provided by a third party (in this case, our process migration layer). API interception techniques have been long used in the PC software industry to enhance operating system capabilities (such as adding compressed file systems). Recently two readily available API interception toolkits have been implemented, one called *Mediating Connectors* [Ba99] (from ISI/USC) and the other called *Detours* [Hu99] (from Microsoft Research). We have has considerable experience building our own API interception packages [Heb99, Nas99], as well as using the Mediating Connectors toolkit [DKK99].

The proposed process migration facility uses techniques that depend on effective use of API interception. The basic idea is that, *as the process executes*, we need to know what files it opens, what graphics devices it uses, what network resources it uses, what it tells the operating system and what the operating system tells the process. Of course, we want to be able to monitor this activity as well as make appropriate changes in the information flowing between the process and the operating system.

We collect this information by intercepting the API calls it makes to the Win32 subsystem and recording the relevant information. With this information (and some other techniques) the process (or parts of it) can be migrated or checkpointed when the need to do so appears. . As the process makes calls to Win32 to display graphical items, we can catch the API call and divert it to another machine for display. File access, network access and most other interactions with the outside world can be handles properly for migratable processes, via API interception.

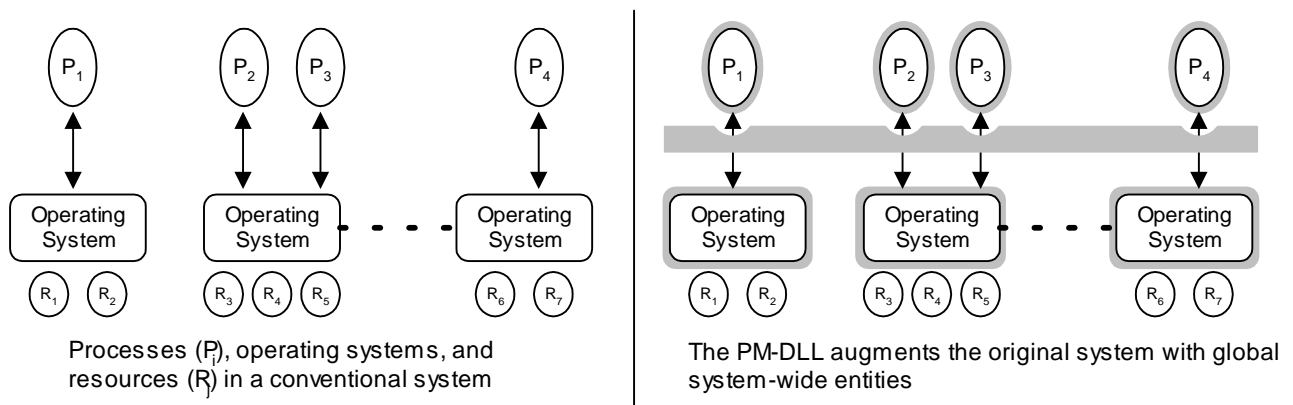
API interception allows us to perform two seeming difficult tasks:

- Providing the application with new facilities without changing the code of the application.
- Incorporate more features into the operating system, without changing the API calls.

4.3. Virtualization

Virtualization is the fundamental idea behind the power of general-purpose process migration mechanisms [B*99B, DKK99]. Consider a user process running a standard application. Under a standard Operating System (OS), this process runs in a logical address space, is bound to a machine, and interacts with the OS local to this machine. In fact, the processes (and their threads) are virtualizations of the real CPUs. Similar virtualizations exist for memory, files, screens, and networks. However, such virtualization is low-level and limited in scope. In our framework, virtualization is defined at a much higher level. *All physical resources* (CPU, memory, disks, and networks) *as well as the OSs* belonging to all the machines are aggregated into a single, unified (distributed) virtual resource space. Thus a process runs on a virtual CPU, uses virtual memory and accesses virtual peripherals (Figure 1).

The application in fact is *enveloped in a virtual shell* (Figure 2). This shell makes the process feel that it is running on a standard OS—Windows NT, in our case. The process interacts with the shell as it would interact with Windows NT. However, the shell creates a virtual world. In general, a running application needs constant access to some set of virtual resources ($V_1, V_2, V_3, \dots V_n$). The real environment contains, in



general, a much larger set of physical resources, ($R_1, R_2, R_3, \dots R_n$). To enable mobility and reconfigurability, we can change the mappings between a virtual resource V_j and a physical resource R_i , *at any time*, as long as V_j and R_i are of the same type. Sometimes it is possible (and desirable) to assign multiple physical resources to a single virtual resource. Of course, substituting a particular file is not possible, but substituting the disk storing the file with another disk is possible.

The underlying techniques make it possible to *virtualize resources* by controlling the mapping between *physical resources* (as visible to an operating system) and *virtual handles* (as visible to running applications). The virtual handles are translated (transparently) to physical handles when an API call reaches the OS. This enables applications to use remote resources as though they are local, and more importantly, it *makes it possible to dynamically change the mapping of virtual to physical resources*. Virtualization not only assigns physical handles to virtual handles, but virtualizes all the identification information. Process ids are replaced by global process ids, IP addresses are replaced by the IP address of a designated gateway machine (see discussion on Network virtualization in Section 7) and so on.

4.4. Implementing Process Migration (The PM-DLL)

The process migration facility relies on a set of mechanisms to provide its services. It consists of a DLL that we call the PM-DLL (or Process Migration DLL). For all processes that are to be migratable, we inject the PM-DLL into the process and create a monitoring thread in the process. This thread then reroutes all API accesses made by other threads in the process, by changing the entries in the DLL import address table. The injection and rerouting can also be done using the Mediating Connectors toolkit [Ba99]. Hence, all relevant Win32 calls are now routed through stubs in the PM-DLL.

After intercepting a call, the PM-DLL does one of the following operations. (1) passes it on to the local Windows NT operating system. (2) passes it to a remote Windows NT operating system. (3) executes it inside the new DLL. (4) executes some code and then passes it to a local or remote Windows NT system.

Simplistic Virtualization: We first present a simplistic design for virtualizing multiple physical machines into a single (virtual) machine. On every machine, every API call made by applications is intercepted and rerouted (using a proxy) to a designated machine M_0 for execution by the OS there. Thus, all the machines run application code, but all the system calls are executed on machine M_0 . In consequence, all applications believe that they are executing on a single machine and, hence, such applications can easily be migrated among physical machines. However, there are shortcomings including performance and lack

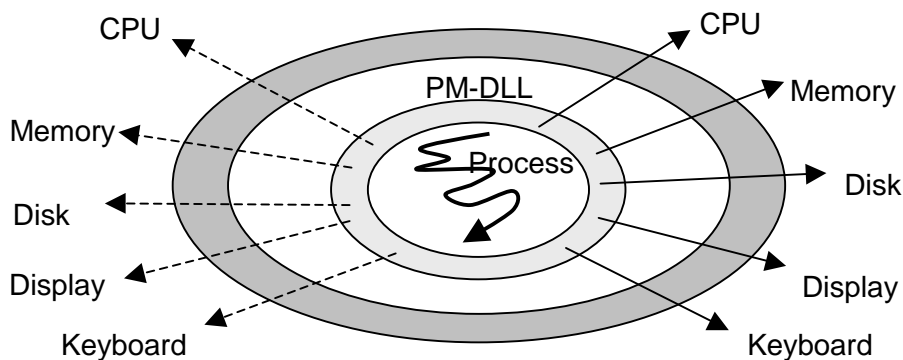


Figure 2: Reassignment of virtual/physical mappings by the PM-DLL

of scalability. For instance, all screen displays appear on M_0 .

The rerouting of system calls should be limited only to calls generated by a migrated process, or for calls requiring remote resource access. Even then, careful design minimizes such remote calls. For instance, system calls to manage processes, such as creating a processes or looking up a process handle, can execute on the machine on which they are generated once the PM-DLL maintains virtual/physical handle binding, as discussed below.

Virtual and Physical Handles: Essential to the implementation of general purpose migration, is the use of virtual handles. For example when a process opens a file, it gets a file handle from Win32. It can use this handle for accessing the file until the file is closed. However, if the process migrates, this handle is useless, as it is a *physical* handle. Hence, when a process opens a file via the PM-DLL, the PM-DLL intercepts this call and stores the returned physical handle but returns to the process a handle, which we refer to as the *virtual handle*. The virtual handle can be used by the process, regardless of mobility, to access that file, due to the transparent translation service provided by the PM-DLL. The virtual handles are used to virtualize I/O connections, sub-processes, threads, files, network sockets, and so on (Figure 3).

GDI/Windowing Virtualization: As an example, consider an interactive application *A*, which reads from a keyboard and writes to a screen. Initially, the application (the process, keyboard, and screen) was executing on a single physical machine M_1 . Then the process is migrated to machine M_2 , the keyboard is mapped to the keyboard of machine M_3 , and the screen is mapped to the screen of machine M_4 . To enable continued execution of *A*, M_3 runs a keyboard input proxy that collects and sends keystrokes to the PM-DLL on M_2 , which delivers it transparently to *A*. Similarly, M_4 runs a screen output proxy, which handles all screen display requests, including those on behalf of *A*, as requested by the PM-DLL on M_2 , following the interception of an API request from *A*.

Thus, the GDI/Windowing Virtualization scheme uses a per-process proxy for every input device located remotely. It uses a per-machine proxy for displaying all output devices routed to that machine. In addition, the per-process DLL for every process sends the GDI requests to the appropriate machine and collects all incoming Windowing events from remote machines. Furthermore, the PM-DLL virtualizes all handles used by an application. Of course, this scheme supports process migration quite well.

File and Network Virtualization: The previous approach, though powerful, needs augmentation. First, some files that the process accessed from M_h may not be accessible on M_t . For such files, M_t opens them using a remote file access proxy, running on M_h .

For network virtualization there are additional concerns.

- **IP Addresses:** Whenever a process asks for an IP address (or host name), it must be provided an *immigration-invariant* IP address. For this, the PM-DLL always returns the IP address of a special *gateway machine*. A process external to the migratable processes can reach a socket created inside the migatable process network, using a proxy running on the gateway machine.

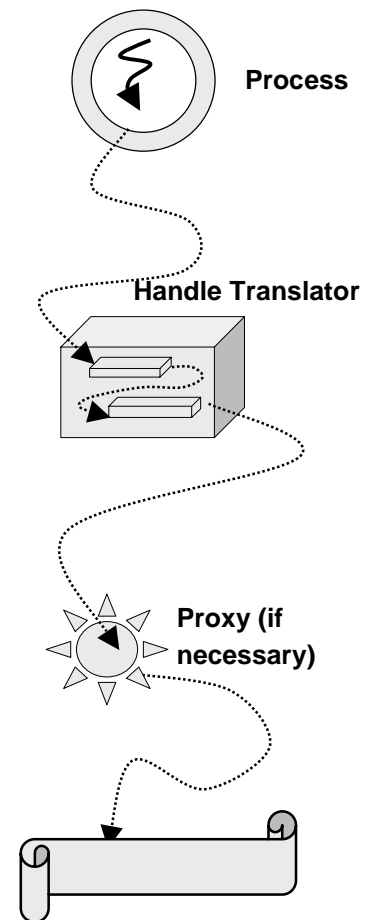


Figure 3: The handle translation mechanism

- **Communication between Migratable Processes:** Suppose P_1 on M_1 and P_2 on M_2 are both using the PM-DLL and are communicating using a socket. P_1 migrates from M_1 to M_3 . The PM-DLL's on the processes on M_1 , M_2 , and M_3 cooperatively close the original socket and open a new socket connecting P_1 to P_2 . In this way, the processes transparently continue communicating in spite of the migration.
- **External Communications:** The above scheme requires that PM-DLLs cooperate to keep virtual sockets open when processes migrate. If P_1 is a migratable process and P_2 is not, then we need to proceed differently. All network connections from migratable processes are routed through a special proxy on a gateway machine. When P_1 opens a connection with P_2 , the connection uses the gateway as a proxy. Now, when P_1 migrates, the P_1 -to-gateway connection is re-established using the above scheme

Process Migration: Once all handles and devices are virtualized, then process migration can be achieved. To migrate a running process, we need to migrate its *image* (*heap* and *stack*) and the *threads* it has created, while maintaining its access to resources, such as *open files* and *network connections*. For simplicity, we consider only two machines for a process, the *home machine* M_h , on which it was created, and the *target machine* M_t , to which it is migrated.

To migrate a process, we need to: (1) start a process on M_t from the same executable as the original process (2) inject the PM-DLL into this process (3) move the data area to this process, (4) move all thread stack to this process (5) load contexts of all threads (5) recreate all handles and (6) start the threads.

The Data Region consists of `.rdata`, `.idata`, `.edata` and `.data` sections. The `.rdata` section consists of read only data like constant strings and constant integers etc. The `.idata` section contains the import data including the import directory. The `.edata` section consists of the export data for an application or DLL. Stacks are that part of the process address space. The size of the stack varies depending on whether the function is entered or exited. When the process is restarted on a new machine, the whole stack space has to be restored. The context of the threads are obtained from machine M_h using the `GetThreadContext()` call and then restored on M_t using the `SetThreadContext()` call. The dynamic memory is transferred by monitoring the `VirtualAlloc()` calls on M_h and calling `VirtualAlloc()` on M_t appropriate number of times.

Similarly, when a process on M_h opens a file, the PM-DLL intercepts the system call, creates a virtual file handle with a global unique identifier, and delivers this virtual handle to the process. A table retains the translation from the virtual handle to the physical handle returned by the OS. Each time the process accesses the file using the virtual handle, the PM-DLL translates it to the local handle and presents it to the OS. After the process migrates from M_h to M_t , the PM-DLL on M_t opens the file, gets a physical handle for it, and associates the original virtual handle with this physical handle. Similarly, socket communication uses translations from virtual to physical handles for sockets.

4.5. API Call Interpretation and Replay

An important component of the virtualization approach is monitoring and possibly intercepting the process interactions with the underlying operating system and vice-versa. These interactions are available to the process migration DLL in the form of a stream of API calls. The main idea of our approach is that since all side-effects in the OS are created using these API calls, the kernel state associated with the process can be recreated on a new node by simply replaying some (all) of the logged API calls.

To take an example, consider a process that executes a sequence of writes against a file that is accessible from all machines on a network. The kernel state associated with this process includes the file pointer

information as well as some state in the file cache. When this process migrates to another node, some of this state needs to be transferred (e.g., the file pointer offset) and some state can be ignored since the process does not rely upon it for correctness (e.g., the cache contents need to be flushed to disk but there is no need to recreate the same contents on the new node). In this case, the PM-DLL needs to determine the value of the file pointer offset: it could have obtained this either by keeping track of the pointer value after each file operation, or could obtain this at migration time by executing an appropriate API call. Similarly, to flush the cache, the PM-DLL could either inject an appropriate API call into the sequence whose effect is to flush the relevant cache blocks, or at the time of migration, it can flush the entire cache.

The above example is representative of the processing of the API call stream in the PM-DLL for different kinds of resource types. In general, the PM-DLL would need to log each API call from the start of execution, perform some computation on its interception, optionally introduce new API calls into the stream, and finally perform additional computation when the return value is being passed back to the application. Given the large number of API calls available at the Win32 interface, a systematic approach is required for figuring out how best to handle the state transfer or recreation of a particular kind of resource. Our goal is to create a framework for API call interception such that starting from a high-level specification of the semantics of this API call and its relationship to other calls in the same group (e.g., all file manipulation calls), it is possible to automatically create the appropriate sequence of operations that must be performed in the PM-DLL to log the appropriate calls, to process them as appropriate, optionally introducing new calls into the stream. Such a framework would simplify the construction of the PM-DLL, as well as serve as a basis for optimizing amongst multiple designs.

The following primitive operations are performed on an API stream by the PM-DLL:

1. **Translation:** Each API call is passed directly to the underlying operating system with optional processing on both the entry and exit paths. For example, virtual-to-physical handle translation is required for API calls referring to virtualized resources.
2. **Forwarding:** Each API call is sent to the source/target node with optional processing on both the entry and exit paths. For example, when only the process display is migrated, all API calls that draw on the screen need to be routed to the target node with accompanying handle translation.
3. **Filtering:** Only a subset of the API calls need be logged. For example, of the GDI API calls that create a graphical window, and draw various figures in it, only the create call need be logged, since the process has a way of accessing the screen contents in its own space.
4. **Summarization:** A sequence of API calls in the stream can be replaced by one or more summary operations (other API calls) with respect to their effect on kernel state. These summary operations can be used both on the source node (to ensure that the kernel side-effects are in a “good” state), and on the target node (to recreate the necessary side-effects). An example of the former is performing an operation that flushes file caches to disk (to ensure consistency), and of the latter is replacing a sequence of file write operations with a seek operation to update the file pointer.

One of the key research problems we propose to explore is understanding which of these primitive operations provides the best support for handling process migration requirements of a particular resource.

5. Status and Experiences

We have been experimenting with API interception techniques for building distributed computing systems for about a year. In the process, we have built several prototypes that can migrate processes from one

machine to another, modulo some constraints. We have successfully moved processes that have active network connections, open files, and are actively displaying graphical output [Heb99, Nas99].

Our results fall into three categories: (1) Network Virtualization, (2) File System Virtualization, and (3) GDI Virtualization. We briefly discuss our progress in each of these categories. We have tested process migration with each of the above mentioned virtualization schemes, but have not yet integrated them in a single package.

Network Virtualization: Using the ideas of handle translation and socket reconnection, we have developed prototypes where a connection between processes *A* and *B* can be broken and then *A* connected to *C*, without process *A* realizing that the process it was talking to has been substituted. From this simple prototype we have advanced to a more complex case where the process *B* is moved to a different machine without either *A* or *B* realizing the movement [Nas99].

File Virtualization: A prototype of file virtualization and migration of processes actively accessing files has been constructed on systems where all machines share a common shared file system. This case does not require any proxies. If the process that is being moved has open file handles, these are closed and reopened at the other machine and the state of the file handle is updated accordingly. The handle translation scheme ensures that any handles internalized in the process do not need to change. File access then continues without any disruptions.

GDI Virtualization: The GDI virtualization testbed works by starting up a regular application (like the “*minesweeper*” game) under the API interception scheme. Then, by sending the GDI API calls to another machine we can display the game on a machine different from where it is running. The mouse movements can also be displayed on the “other” machine using a Windows mechanism called “hooks”. We have then experimented with moving the process without moving the display and have been successful at that. We have recently become aware of a commercial product on Windows NT that provides similar functionality.

We have also experimented with an alternative design, relying on the explicit injection of a single thread that handles all displaying. Such applications can be migrated by moving only the shared data of the process over to the new machine and restarting the display thread from the beginning (without restoring the state of the thread). Note that the injection itself can be achieved transparent to the application using API interception techniques. This technique is simple and circumvents the need for replaying GDI events. We have to experiment to see if this technique can be used for general GUI based applications. Since the GUI thread is context-less, we feel we can use this technique for most Windows applications [Heb99].

6. Related Work

We have proposed a user-level virtualization approach based upon API interception techniques for capturing process interactions with its environment, enabling state transfer and recreation for process migration. Our approach is related both to previous works that address transparent mobility of computations using kernel-level support, and to those that leverage API interception techniques to extend OS functionality.

Network file systems such as NFS [NFS], AFS [Ka99], and Coda [S*90] provide location-independent access to files. Distributed operating systems such as Chorus [R*92], MOSIX [B*93], Amoeba [T*90], SPIN [Be*95], and Sprite [DO91] rely on extensive kernel modifications to support mobility. While these systems can support migration transparent to the application and with minimal residual dependence, these gains come at the cost of significant OS complexity.

API interception techniques have recently received widespread attention in the context of both Unix and Windows NT systems. Of the former, the Interposition Agents [Jo93] and Ufo [A*98] systems provide seamless access to distributed files, addressing only part of the general problem of process migration. Of the latter, NT-SwiFT [Hu*98] uses these techniques for providing fault tolerance using checkpoint/restart for regular, unmodified applications on top of NT. However, NT-SwiFT only captures some of the process' interactions with the underlying OS, focusing on the restricted problem of restarting a computation on the same machine where it was checkpointed.

More closely related are systems such as GLUnix [G*98] and DUNE [CP99] that rely on user-level mechanisms for hiding the location of system services and allowing mobility of computations. GLUnix differs from our approach in that it moves OS functionality into user level libraries, requiring the user applications to re-link with these libraries before they can take advantage of these features. DUNE shares the same objective as our project—transparent mobility of computations, but focuses on using process migration to improve throughput/performance of distributed computations in a networked environment. In contrast to DUNE, which transparently but always maintains residual dependencies on the source node using forwarding, our approach articulates the design of more general API logging and proxy strategies that minimizes the need for such dependencies.

7. References

- [A*98] A. D. Alexandrov, M. Ibel, K. E. Schauer and C. J. Scheiman, Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System, *ACM Transactions on Computer Systems*, August 1998.
- [B*93] A. Barak, S. Guday and R. G. Wheeler, The MOSIX Distributed Operating System, *Lecture Notes in Computer Science*, Vol. 672, Springer, 1993.
- [B*99A] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *Future Generation Computer Systems*, 1999.
- [B*99B] A. Baratloo, P. Dasgupta, V. Karamcheti, and Z.M. Kedem, Metacomputing with MILAN, *Heterogeneous Computing Workshop, International Parallel Processing Symposium*, April 1999.
- [Ba99] R. Balzer, Mediating Connectors, *ICDCS Middleware Workshop*, 1999.
- [Be*95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers and C. Chambers, Extensibility, Safety and Performance in the SPIN Operating System, *15th Symp. on Operating Systems Principles*, December 1995.
- [C*95] J. Casas, D. L. Clark, R. Konoru, S. W. Otto, R. M. Prouty, and J. Walpole, MPVM: A Migration Transparent Version of PVM, *Computing Systems: The Journal of the USENIX Association*, 8(2), Spring 1995.
- [CP99] J. Cruz and K. Park, Towards Performance-Driven System Support for Distributed Computing in Clustered Environments, *J. of Parallel and Distributed Computing* (to appear), 1999.
- [D*94] L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot, DynamicPVM: Dynamic Load Balancing on Parallel Systems, In W. Gentsch and U. Harms, editors, *High Performance Computing and Networking*, pp. 273-277, April 1994, Springer Verlag, LNCS 797.
- [DKK99] P. Dasgupta, V. Karamcheti, and Z.M. Kedem, Transparent Distribution Middleware for General Purpose Computations, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, June 1999.
- [DO91] F. Douglass and J. Ousterhout, Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software—Practice and Experience*, 21(8), 1991.
- [G*98] D. R. Ghormley, D. Petrou, S. H. Rodrigues and A. M. Vahdat, GLUnix: A Global Layer Unix for a Network of Workstations, *Software—Practice and Experience*, 28(9), 1998.
- [Heb99] R. Hebbalalu, *File I/O And Graphical I/O Handling for Nomadic Processes on Windows NT*, MS Thesis, 1999. Arizona State University.
- [Hu*98] Y. Huang, P. E. Chung, C. Kintala, D. Liang, and C. Wang, NT-SwiFT: Software Implemented Fault Tolerance for Windows NT, *2nd USENIX Windows NT Symposium*, July 1998.

- [Hu*99] G. Hunt and D. Brubacher, Detours: *Binary Interception of Win32 Functions*, Microsoft Research Technical Report, MSR-TR-98-33, February 1999
- [JHB88] E. Jul, H. Levy, N. Hutchinson, and A. Black, Fine-grained Mobility in the Emerald System, *ACM Transactions on Computer Systems*, 6(1), 1988.
- [Jo93] M. B. Jones, Interposition Agents: Transparently Interposing User Code at the System Interface, *14th Symp. on Operating Systems Principles*, ACM Press, December 1993.
- [K*96] Y. A. Khalidi, J. M. Bernabeu, V. Matena, K. Shirriff, and M. Thadani, Solaris-MC: A Multi Computer OS, *1996 USENIX Conference*, January 1996.
- [Ka99] M. L. Kazar, Synchronisation and Caching Issues in the Andrew File System, *1988 USENIX Conference*, Winter 1988
- [LLM88] M. Litzkow, M. Livny, and M. Mutka, Condor—A Hunter of Idle Workstations, *8th International Conference on Distributed Computing Systems*, 1988.
- [MSD99] D. McLaughlin, S. Sardesai, and P. Dasgupta, Preemptive Scheduling for Distributed Systems, *11th International Conference on Parallel and Distributed Computing Systems*, 1998.
- [Nas99] R. Nasika, *Migration Of Communicating Processes via API Interception*, MS Thesis, 1999. Arizona State University.
- [NFS] Sun Microsystems, Inc., *NFS: Network File System Protocol Specification*, IETF Network Working Group RFC 1094, March 1989.
- [R*92] M. Rozier, V. Abrossimov, F. Armand, M. Gien, M. Guillemont, F. Hermann, and C. Kaiser, Chorus (Overview of the Chorus Distributed Operating System), *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.
- [S*90] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, Coda: A Highly Available File System for a Distributed Workstation Environment, *IEEE Transactions on Computers*, 39(4), April 1990.
- [SD98] S. Sardesai and P. Dasgupta, Chime: A Windows NT based parallel processing system, *USENIX Windows NT Symposium*, Seattle, August 1998. (Extended Abstract).
- [SMD98] S. Sardesai, D. McLaughlin and P. Dasgupta, Distributed Cactus Stacks: Runtime Stack-Sharing Support for Distributed Parallel Programs, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, July 1998.
- [Sun90] V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, *Concurrency—Practice and Experience*, 2(4), December 1990.
- [T*90] A. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen, and G. van Rossum, Experiences with the Amoeba Distributed Operating System, *Communications of the ACM*, 33(12), 1990.