Transparent Distribution Middleware for General Purpose Computations[1]

Partha Dasgupta[†], *Vijay Karamcheti*[‡] *and Zvi M. Kedem*[‡]
[†]*Arizona State University*
[‡]*New York University*

Abstract

A huge installed base of general-purpose (often sequential) applications cannot take advantage of distributed systems. This paper presents a design and initial prototypes of a set of techniques, called adaptive virtualization, that endows regular applications with features for exploiting a distributed environment. These features include the ability to be scheduled on diverse CPUs, adapting to availability of resources, handling faults, and moving around in a distributed system.

Our approach is to first lift the executing binary of the application off of the operating system using the API interception technique. Then we insert a middleware layer between the application and the operating system. This layer emulates the base operating system as far as the application is concerned, but actually builds a distributed operating system that the application run on. This technique essentially virtualizes all resources of a distributed system and creates a large virtual multiprocessor.

Keywords: Distributed Operating Systems, Middleware, Process Migration.

1. Introduction

Since mid 1980's there has been extensive work on distributed computing, distributed (or decentralized) programming, distributed operating systems, and distributed applications. However, these efforts have not achieved widespread success in transforming networked machines into easily useable, reliable distributed computing platforms.

A major obstacle that has proven difficult to surmount is the *application development barrier*. When a new system with new features is built, it comes with an API and a set of methods to build applications. Older applications do not run on the new system, nor can they take advantage of the new system's capabilities.. Since developing new applications using a new development environment is difficult, the programmers shun the system, and it gets only limited acceptance.

In this paper, we present a novel method of middleware development, which does not suffer from the excruciating problem of having to redesign, re-code, or even recompile applications. All *binaries* of all existing applications that run on a standard operating system (OS) run on the new, distributed platform. Binary compatibility with all applications, gives us the "no new API" method of system development.

Our approach takes a base OS (Windows NT in our case), *and* all the applications that run on it, and transparently converts that system into a "different" one. This "different" system is called a *"Computing Community"* or *CC*. A CC has features that are selected when the CC is constructed. For example, a CC that can have the features of *transparent distribution*, *global scheduling, fault tolerance*, and *application adaptation*.

2. The Mechanism: API Interception

The basic, low-level mechanism used to make it possible for any application to use the facilities of a distributed system is *API interception*. An application runs on top of an operating system. It uses the API of the operating system to obtain services of the operating system. API interception a method of unobtrusively and transparently intercepting all or part of the API calls the application makes to the operating system and then emulating the function

[1] This research is partially supported by grants from DARPA/Airforce Research Laboratory – Rome Research Site (F30602-96-1-0320), NSF (CCR-94-11590 and CCR-95-05519), Intel, and Microsoft.

provided by the native API call using additional software.

API interception allows the application to be *"lifted-off"* of the operating system and a layer of software inserted between the application and the operating system. This software is a middleware layer that is invisible to the application, but endows the application with features it was not designed for. From a different perspective, we can say this middleware enhances the base operating system into a new operating system that has features above and beyond the original operating – system, and still is binary compatible with the original operating system.

Our research project is using the API interception mechanism to build a distributed operating system that will provide all application with features of distribution without the need for reprogramming or even re-compilation. There are several toolkits available to automate the task of API interception. We are currently using *the Mediating Connectors* toolkit from ISI [Ba99] A toolkit with similar functionality called *Detours* [Hu*99]. is available from Microsoft. The distributed system we are building is called the Computing Community.

3. The Computing Community (CC)

Definition: A CC is an aggregation of computing and information resources—and even CCs—*all drawn from diverse sources*. CCs are dynamic and hierarchically constructed. CCs are *self-repairing*, and continue delivering, *adequate,* and *predictable performance of the key services despite faults and other imperfections of the execution platform.*

Effectively, a CC is a single, dynamically changing, virtual multiprocessor system, physically built of many components. The physical network disappears from the view of the computations that run on the CC.

The CC comprises three synergistic components: (1) Virtual Operating System (2) Global Resource Manager, and (3) Application Adaptation. These components are responsible, respectively, for providing a unified view (virtualization) of CC resources and privileges, integrating diverse components into the CC, and adapting to changes in CC resource characteristics.

1. **The Virtual Operating System** (**VOS**) is a layer of software that non-intrusively operates between the applications and the standard operating system. The VOS presents the standard Windows NT API to the application, but can execute the same API calls differently.
2. **The Global Resource Manager** component manages all CC resources, dynamically discovering the availability of new resources, integrating them into the CC, and making them available for use by CC computations.
3. **The Application Adaptation** component enables the computations to take full advantage of CC resources and provides self-repair capabilities.

Figure 1 shows a conceptual view of a CC. It takes a set of operating systems, and a set of resources, and via a layer of middleware converts it into an integrated system. A computation executing within a CC has controlled and customizable access to the union of the resources and privileges contained in the CC. CCs can expand and contract dynamically, and the computations are completely mobile within CCs.

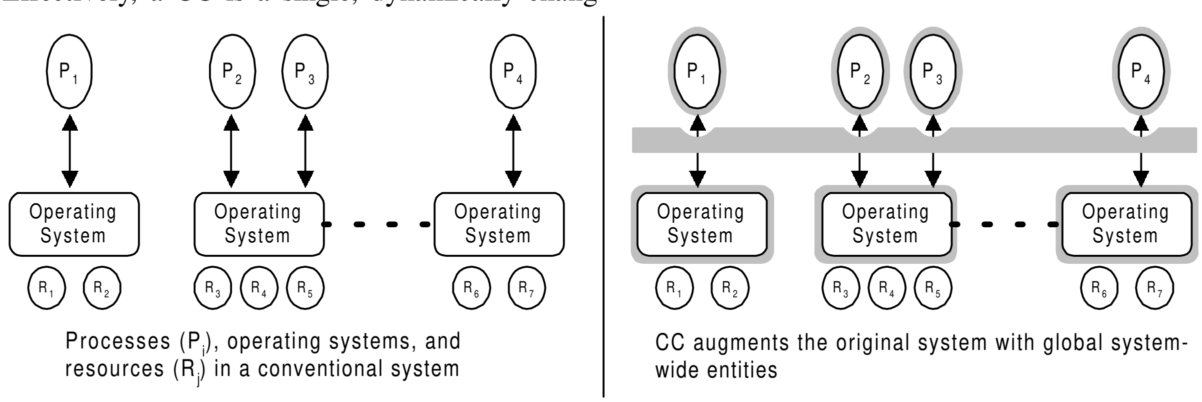

Figure 1: From a standard operating system to a Computing Community.

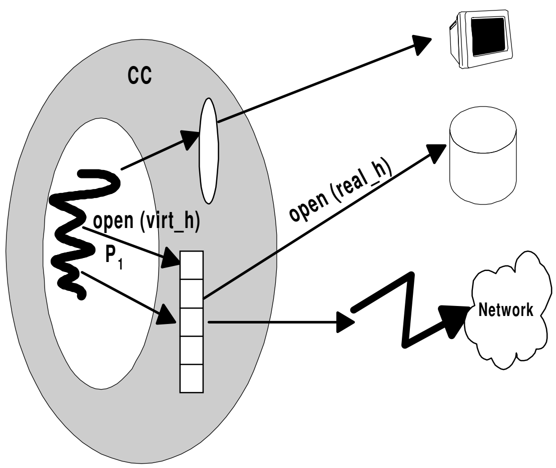

Figure 2: The interaction of a process with the virtual shell.

This paper describes that part of the middleware that makes the transparent distribution possible, namely the Virtual Operating System (VOS).

4. Virtual Operating System Design

4.1 The Virtualization Concept

Virtualization is the fundamental idea behind the power of a CC. Consider a user process running a standard application. Under a standard 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 the CC framework, virtualization is defined at a much higher level. *All physical resources* (CPU, memory, disks, and networks) *as well as the OS*s belonging to all the machines are aggregated into a single, unified (distributed) virtual resource space.

As in an operating system, a process in the CC runs on a virtual CPU, uses virtual memory and accesses virtual peripherals. However, in addition, it 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 made of the aggregate of the physical worlds in the CC.

In general, a running application needs constant access to some set of virtual resources (V_1, V_2, V_3, ... V_n). The CC contains, in general, a much larger set of physical resources, (R_1, R_2, R_3, ... R_n). To enable mobility and reconfigurability, the CC 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, as described below. Of course, substituting a particular file is not possible, but substituting the disk storing the file with another disk is possible.

Here are a few of the advantages:

- The users can move their virtual "home machines" at will, even for applications that are currently executing. This is the *ultimate* mobile computing scenario.
- A critical service running on machine M_1 can be moved to machine M_2 if M_1 has to be cast away. This is essential for performing maintenance without service interruption.
- Schedulers can control the complete set of resources. This is important for providing high performance.
- The provision of multiple physical resources for a single virtual resource delivers important new capabilities ranging from duplicating application displays on multiple screens to replicating processes for fault-tolerance (all without programming the application to do so).
- Applications can use resources, transparently aggregated from several machines. For example, a memory-intensive application can use memory in remote machines.

To summarize, the CC presents a single machine interface to all applications, although they execute on different physical machines.[ZMK1]

4.2 The Virtual Operating System (VOS)

The Virtual Operating System (VOS) is a layer of software that runs on *every* machine in the CC, shielding running applications from the static nature of the standard OS. The VOS runs at the user level, and extends the capabilities of an OS by monitoring and *intercepting* system calls made by all application running on that machine. Conse-

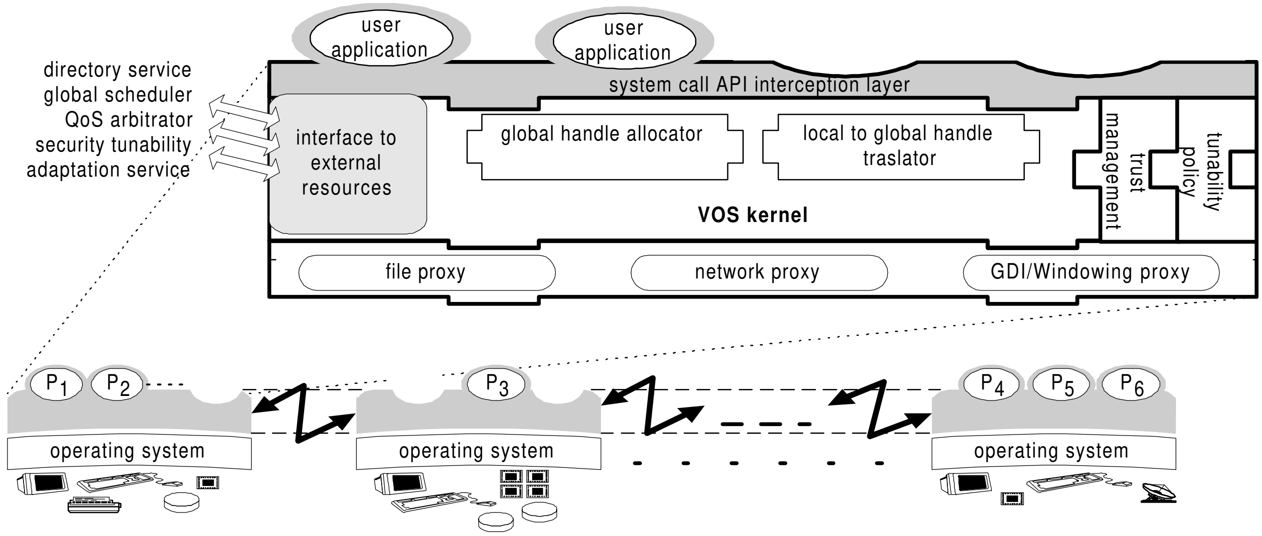

Figure 3: The components of the Virtual Operating System.

quently, the behavior of intercepted system calls can be modified while maintaining the original API, and OS capabilities can be extended by dynamically "installing" additional functionalities.

The underlying techniques implemented by VOS 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 (via the VOS). 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*.

The VOS, provides the following major functions:

- It creates a "unified" virtual machine interface for application processes. This allows processes running on the CC to see all machines as a single (virtual) Windows NT machine supporting the Win 32 API, thus providing full binary compatibility with Windows NT.
- It provides the low-level mechanism for process migration and dynamic resource allocation. It uses the global resource manager and application adaptation module for process reconfiguration, adaptation, and global resource management inside the CC.

5. The VOS Implementation

The VOS relies on a set of integrated mechanisms to provide its services. It consists of a set of DLLs and a set of proxy processes. Some of the DLL's are per process, i.e. each process has a copy of such DLL.[2] Some of the DLL's are per-machine, i.e. all processes on a machine share the same DLL. The same is true for the proxies, with some being per-process and some per-machine. All the components of the VOS are described below and are shown in Figure 3.

5.1 Simplistic Virtualization and Handles

The VOS works by intercepting all calls made from the application to the Win32 interface. The VOS intercepts API calls by injecting a VOS DLL into a running process, creating a monitoring thread in the process, and then rerouting Win32 API calls by changing the DLL import table in the process. After intercepting a call, the VOS 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 VOS. (4) executes some VOS 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

[2] Technically applications share a single DLL, but the presentation is easier if phrased in terms of multiple DLL's.

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. For instance, all screen displays appear on M_0 and this architecture is, of course, not scalable.

However, 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 VOS maintains virtual/physical handle binding, as discussed below.

Virtual and Physical Handles: Equally essential 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 on top of a VOS, the VOS intercepts this call and stores the returned physical handle but returns to the process a handle, which we refer to as *virtual*. 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 VOS. The virtual handles are used to virtualize I/O connections, sub-processes, threads, files, network sockets, etc.

5.2 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*, the VOS on M_3 runs a keyboard input proxy that collects and sends keystrokes to the VOS 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 VOS 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 per-process DLL virtualizes all handles used by an application. Of course, this scheme supports process migration quite well.

5.3 Process Migration

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.

When a process makes a system call creating a thread (while still on M_h), the VOS intercepts the calls and records the created thread. Thus, after migration to M_t, the VOS can recreate the threads and their contexts. Similarly, when a process on M_h opens a file, the VOS intercepts the system call, creates a virtual file handle with a CC-wide 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 VOS translates it to the local handle and presents it to the OS. After the process migrates from M_h to M_t, the VOS 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.

5.4 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 VOS always returns the IP address of a special *CC-gateway machine*. A process external to the CC can reach a socket created inside the CC, using a proxy running on the gateway machine.

Intra-CC sockets: Suppose P_1 on M_1 and P_2 on M_2 are communicating using a socket. P_1 migrates from M_1 to M_3. The VOS's 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.

Inter-CC sockets: The above scheme requires VOS's that cooperate. If P_1 is executing on a CC and P_2 is not, then we need to proceed differently. All routing of inter-CC-networking will use the 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 intra-CC-networking scheme

6. Status and Experiences

Computing Communities is a large, ongoing research project that is expected to take years from now to reach completion. However, we have obtained significant early results of attaining virtualization of operating system resources. As stated earlier, we are using the *Mediating Connectors* [Ba99] toolkit to automate the API interception mechanism. Our results can be divided into four categories:

1. GDI Virtualization
2. Network Virtualization
3. File System Virtualization
4. Process Migration

In this section, we discuss the progress in each of these categories, in brief.

6.1 GDI Virtualization

The GDI virtualization testbed works by starting up a regular application (like clock) under the API interception scheme. Then, by sending the GDI API calls to another machine we can display the clock on a machine different from where it is running. We have then experimented with moving the process without moving the display and have been successful at that. Currently we can migrate processes while they are actively displaying window contents.

6.2 Network and File Virtualization

Similar to GDI Virtualization we have developed prototypes where a connection between processes *A* and *B* can be broken and then *A* connected to *C* without 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.

If the process that is being moved has open file handles these are closed and reopened at the other machine, and the handle translating scheme ensures that the handles do not change. File access then continues without any disruptions.

6.3 Process Migration

All the above prototypes involve migrating processes. The process space under NT is composed of the *text* region, *.bss* region, *.rdata* region and the *.data* region. To achieve process migration, the thread in the process is suspended and its context extracted. Then all but the text region is transferred to another thread, then the context is restored, and the thread resumed. Of course, all the handle translation and the resource acquisitions have to be done at this point. We have tested process migration with each of the above mentioned virtualization schemes, but have not integrated all of them yet.

7. Related Work

Computing Communities (CC) enable standard applications running on top of commodity operating systems to transparently acquire the benefits of secure, dynamically reconfigurable, survivable computing. Virtualization traditionally has dealt with location transparency and process mobility. Traditional solutions have required modifications to either the application programs or the underlying operating system. For instance, GLUnix [Gh*98] is a set of programs and library functions to provide applications running on a network of Unix workstations with a uniform view of distributed resources. In other projects, such as MOSIX [Ba*93] and SPIN [Be*95], operating systems have been modified to support distributed computations and to provide transparent process mobility.

More recent approaches have relied on extending standard operating systems, intercepting operating system calls made by applications, and manipulating these calls to provide a richer set of functionalities than originally intended. Interception techniques, first proposed in Interposition Agents [Jo93], have been successfully used to create a user-level sandbox called Janus [Go*96], build a virtual file system Ufo [Al*98], and provide transparent checkpoint/recovery service in NT-Swift [Hu*98].

8. Conclusions

We have outlined the design and key aspects of a preliminary implementation of middleware which provides a flexible platform for executing general purpose applications on distributed environments. The applications "believe" that they are running on a standard operating system, Windows NT in our project, but in fact are transparently managed so that they can be provided system and other resources and migrated mostly without their knowledge to provide better reliability and end-to-end functionality for the users.

9. References

[Al*98] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser and Chris J. Scheiman, *Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System,* ACM Transactions on Computer Systems, 16(3), pp. 207-233, August 1998.

[Ba*93] Amnon Barak, Shai Guday and Richard G. Wheeler, *The MOSIX Distributed Operating System*, Lecture Notes in Computer Science, Vol. 672, Springer, 1993.

[Ba99] Robert 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,* Proceedings of the 15th Symposium on Operating Systems Principles, pp. 267-284, December 1995.

[Gh*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), p. 929, 1998.

[Go*96] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer, *A Secure Environment for Untrusted Helper Applications,* Proceedings of the 6th Usenix Security Symposium, July 1996.

[Hu*98] Yennun Huang, P. Emerald Chung, Chandra Kintala, De-Ron Liang, and Chung-Yih Wang, *NT-SwiFT: Software Implemented Fault Tolerance for Windows NT,* 2nd USENIX Windows NT Symposium, July 1998.

[Hu*99] Galen Hunt and Doug Brubacher, Detours: Binary Interception of Win32 Functions, Microsoft Researach Technical Report, MSR-TR-98-33, February 1999

[Jo93] Michael B. Jones, *Interposition Agents: Transparently Interposing User Code at the System Interface,* Proceedings of the 14th Symposium on Operating Systems Principles, pp. 80-93, ACM Press, December 1993.

Sponsor Acknowledgment: Effort sponsored by the Defense Advanced Research Projects Agency and AFRL/Rome, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.
Sponsor Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, AFRL/Rome, or the U.S. Government.