Task Management Issues in Distributed Systems

Ahilan Anantha, Maki Sugimoto, Andreas Suryawan, Peter Tran

University of California, San Diego

November 21, 1998

Abstract

One of the main goals of distributed systems is allowing idle processing resources to be utilized. To accomplish this, there must be mechanisms to distribute tasks across machines. We examine the task management mechanisms provided by several distributed operating systems, and analyze their effectiveness.

1 Introduction

A major motivation for constructing a distributed operating system is to perform coordination of decentralized resources in order to raise the utilization of the system as a whole. It is through the management of tasks that a system is able to optimize the parallelism being offered, thereby increasing utilization.

There are quite a few interesting attributes of distributed operating systems, and notable techniques used in handling these. We examine the following attributes and techniques of task management:

1. Ownership of CPU resources
2. Homogeneous vs heterogeneous environment
3. Remote execution/process migration
4. Namespace transparency
5. Load information and control manager

Design choices made in the systems surveyed reflect a set of tradeoffs considered by the system architect: complexity, residual dependencies, performance, and transparency. [7] We analyze how each operating system copes with these conflicting factors to provide efficiency and maintainability. And finally, we consider improvements to these systems.

2 Techniques and Attributes of Task Management

2.1 Ownership of CPU Resources

The operating systems we discuss in this paper fall into two basic classes of environments: 1) those where machines are "owned" by particular users, such that a machine's processing may be used only when the owner is not using it, and 2) where there is notion of ownership of machines, all processors are available for use by all users. Operating systems of the first class are typically designed for environments of graphical workstations.

The Sprite and Condor operating systems are designed for the lab graphical workstation environment. There is no separate CPU server cluster, the workstations themselves make up the distributed system. A user is expected to interact with the operating system by way of a windowing system, which is a highly CPU and memory intensive interactive process. Interactive processes have randomly fluctuating loads because they act in response to user activity, which itself is randomly fluctuating. At the same time, interactive processes have minimum delay requirements because users require real-time response. Windowing systems pose an even greater problem because they require a large percentage of processing power, while traditional text mode interaction is fairly lightweight. In order to satisfy these delay requirements, it becomes necessary to reserve the maximum amount of CPU resources that a windowing system would require.

In non-distributed systems, the user of a graphical workstation is expected to actively control which processes may run on the system to satisfy his delay requirements. If the user of a graphical workstation has ownership of all the user processes that can hog the system resources, he can suspend or terminate the processes that prevent the useability of the console. However, if other users were permitted to easily run processes on remote systems, the console user will lose the ability to control the interactive response time.

For this reason, these operating systems give a second class status to remote processes. Remote processes are only allowed to utilize the resources of a workstation if the workstation is not already busy serving its console user. The CPU resources can be taken back from remote processes if the console user desires them. As such, the console user can be considered the owner of a workstation's CPU resources.

Sprite and Condor will only permit remote processes to run on a system when the system is idle and will evict remote processes once a user starts using the console.

The other class of distributed systems consists of environments of dedicated CPU servers, data servers, and graphical terminals. The bulk of the processing power in these environments is contained in the CPU server. The computers with graphical displays are essentially graphical terminals, they have sufficient processing ability to run the windowing system processes but require no more. All other CPU intensive processes are executed remotely on a CPU server. No user "owns" the CPU server, every user gets a guaranteed share of its resources. Conversely, no remote processes would be allowed on a graphical terminal. The work of trying to determine whether a graphical workstation is idle is unnecessary, since the graphical terminals would have minimal processing resources to offer. All the resources of a graphical terminal can thus be reserved for the windowing system.

Clouds, Alpha, MOSIX, Plan 9, and Solaris MC fall under this category. Solaris MC provides a process migration mechanism for server machines, allowing processes to be migrated across server machines in cases where one server must be brought down for maintenance. In these cases, migrated processes can be permitted to be inefficient and to tax the resources of the hosting system. But the necessity of maintaining services across server disconnections outweighs these factors. Therefore Solaris MC suggests the need for a distinction between servers (which are prepared to offer the resources to accept migrated processes) and user workstations (which are not willing to accept the burden of migrated processes). Clouds, Alpha, and Plan 9 are alike in that they consist of separate highend CPU and data servers. A MOSIX system consists of a large number of commodity workstations, all of which may play an equal part in serving data and processing.

2.2 Remote Execution

Remote execution and process migration are the techniques used in distributed systems to share CPU resources. Remote execution is the ability to create processes on remote machines. Process migration is the ability to relocate processes between nodes in mid-execution.

Plan 9 supports remote execution on CPU servers explicitly specified by the user. Processes cannot be migrated, therefore remotely executed processes spend their entire lifespan on the remote CPU server, from creation to termination. Condor also supports remote execution only.

Sprite supports remote execution through the mechanism of process migration. A request for remote execution of a new process on Sprite would need to be reduced to creating the process locally, and attempting to migrate the process soon after. In the case of an immediate remote execution, local memory need not be allocated for the code and data if an idle computer is available to start with. Sprite, however, doesn't provide this optimization. All execution begins locally.

Process migration is only a request that an application can make. Processes are only permitted to execute on remote workstations that are idle. If there are no idle machines then the request may be denied and the process would continue to execute on the local machine. Therefore, a process cannot also be expected to be able to begin execution on a remote system either.

When the computer ceases to be idle, all remote processes must be evicted. Therefore, every process must have the notion of a home machine, which is the machine from which the user invoked the process.

Solaris MC provides remote execution and process migration mechanisms. User must explicitly call the *rexec* system call to carry out remote execution. Unlike Plan 9, the destination node need not be specified. Process migration is assumed to be used mainly for offloading processes from a node being shutdown for mainteanance.

MOSIX is the only oprating system that makes use of process migration for the purpose of load-balancing. Any user process can be migrated any time to any available node transparently.

2.2.1 Thread Migration in Object Based Systems

Object-based distributed systems have a different way of organizing resources by representing them with passive objects. Objects encapsulate code and data. An object's code is executed using a procedural interface called invocation. Objects are large-grained in that they have their own virtual-address space, and there is relatively large overhead with the invocation and storage of an object. For these reasons, objects generally implement storage and execution of large-grained data and programs.

Clouds is an example of an object-based distribute operating system. A thread in Clouds is a path of execution made up of a series of calls to object methods. Each call is referred to as an invocation that the object responds to. An object by itself is passive. When a thread invokes an object's method, the thread enters that object's virtual-address space and begins execution. Each invocation of a method is called a segment of the thread that invokes it. This segmentation of threads is how Clouds provides distributed execution.

Note how this is different from the traditional model of processes and process migration. There, a process executes within one virtual-address space unless it is migrated to another node. Migration is expensive, and it is expected not to occur more than once or twice.

In Clouds, migration takes place with object granularity. That is, as a thread proceeds, it may invoke objects on different nodes. A thread's path of execution ncessarily crosses through all of these nodes. It is strategic placement of objects that would be used as a mechanism for load balancing.

Since there are no address space associated with each of the threads, objects on remote or local node can be invoked with the same semantics. Threads can cross node boundaries with the minimum penalty of network overheads.

2.3 Namespace Transparency

A required feature of distributing systems is hiding from a process the fact that it is executing remotely or locally. This transparency should also be maintained with regard to the user. The user should be able to interact with the process in the same way (as the local case) regardless of where the code is executing.

Maintaining this transparency requires changes to the traditional operating system model. Transparency refers to the distributed operating system giving each process a single, uniform view of resources, including the filesystem and I/O devices, regardless of which computer it is running on. The design and implementation of the space of accessible resources, or namespace, directly affects the management of the distribution of processes.

Many operating systems achieve this transparency by the enforcement of a uniform, global namespace. The filesystem will appear the same to every process on every node. One solution follows from mounted filesystems in traditional UNIX. A namespace is constructed as a union of mounted file systems.

Object-based distributed systems provide a different solution. There is no notion of a traditional filesystem, only objects. Resources are encapsulated in objects. Naming takes place with object granularity. This provides a flat namespace. At the system level, all objects are identified by a globally unique bit string. A user-level name service is provided to translate user-registered names to system-level names.

Sprite employs the uniform global name space model. File servers provide domains, similar to UNIX filesystems, that are mounted as subdomains of each other with one domain selected as the topmost, root, domain. This view of the domain hierarchy is the same for every computer in the cluster. This can be contrasted with Sun's NFS, where every client may choose the local mount point of a remote filesystem. In Sprite, the remote file server decides the mount point all the clients must use. Among other advantages, this guarantees that every file in the distributed filesystem has a single globally defined pathname. This makes it possible to migrate programs that attempt to manipulate files. [7]

Solaris MC also employs the uniform global namespace model. The Solaris MC file system, which is built on top of the existing Solaris file system, interposes all file operations and forwards them to the server where the file actually resides. Any process can open a file located anywhere in the system using the same pathname, thus allowing programs to be located on arbitrary nodes. [4]

The object-oriented semantics of Clouds provides a flat namespace along with global accessibility (any thread can reference any object). This is also essentially Clouds' mechanism for distributed shared memory. Only through invocation is access to an object's data allowed; input and output parameters are pass-by-value only. This protects the internal environment of an object. Capabilities-based protection is provided for controlling global accesses to objects. [1]

Plan 9 has an interesting policy for managing name spaces. Every client process can have a local namespace, which have the same semantics of a localized filesystem interface. User-level servers in Plan 9 have the ability to "export" filesystem interfaces to their clients. In fact, these exported filesystem interfaces are the primary means for which Plan 9 servers export all their resources. Some of the objects in these name spaces may refer to globally distinct files in the distributed filesystem, but some may refer to a local copy of a global resource.

For example, the same global Plan 9 filesystem can be used by clients of different processor architectures. A user on different systems may refer to a binary executable using a common pathname, such as /bin/date, but the actual binary file that is utilized will depend on the processor architecture. Devices stored in /dev will refer to devices in the local name space. Some of these devices may refer to actual kernel recognized devices, or they may refer to pseudo device interfaces which user-level servers export. For example, a window in the Plan 9 windowing system exports the devices /dev/mouse, /dev/bitblt, and /dev/cons (which refer to the mouse, bitmapped display interface, and character mode console interface). Each window will export the same devices in their name spaces, but the actual device files are local copies of the pseudo devices exported by the Plan 9 windowing system. The windowing system will multiplex accesses to the actual physical devices.

To support the ability to run processes on remote servers, and have them appear to be running locally, Plan 9 provides the ability to export the local name space to a remotely executing process. The remotely executing process will then have the same view of the filesystem as it would if it had been executing locally. And it would have access to the same devices (real or virtual) as on the local system, because these would also be exported as part of the local name space. [6]

2.4 Homogeneous vs Heterogeneous Environments

Many distributed operating systems can be run on heterogeneous environment. However, all of these operating systems that allow process migration have instated the requirement that all computers involved in process migration have the same processor architecture.

The primary obstacle to heterogeneity is that the execution state of a process is highly architecture dependent. When the source and destination systems are of the same architecture, the code and data segments, registers, stack, and heap can simply be copied without any changes. With differing processor architectures, all of these might need to be significantly modified. Such modification is likely to be expensive and will add significant complexity to the system.

The operating systems we've discussed that support process migration (MOSIX, Sprite, Solaris MC) have the requirement that all machines accepting migrated processes be of the same processor type.

Clouds' distributed execution model also does not explicitly support heterogeneity. Objects on the data servers are stored in a single machine language, so heterogeneous CPU servers would require the machine code be converted from one language to another. This is a complication that would break the symmetry of the Clouds system, and would be expensive to carry out.

Many of these operating systems will permit a data server to be of a different processor architecture, since migration would never take place there.

Plan 9's CPU servers can be heterogeneous. Each program is compiled beforehand for the architecture it intends to be executed on. This prohibits the implementation of process migration in Plan 9.

2.5 Load information and control manager

Distributed operating systems, by their nature, pool processing resources together. Access to common processing resources must be mediated by some entity or entities. The determination of which process will execute on which processor we term task distribution management, and the entities that make this determination we term task distribution managers. Task distribution decision making may be centralized onto one manager or decentralized onto many managers.

One disadvantage of centralized management is its inherent inscalability. The overhead associated with maintaining all the load information and making choices among all the nodes grows with the number of nodes. Another disadvantage is that the failure of the central node brings down the whole mechanism.

We can decentralize this decision making by giving a number of nodes the ability to act as task distribution managers. Each manager would control a partition of the nodes in the system. In this configuration, each managing node essentially becomes the central manager for a smaller distributed system [9]. It can make its own decisions to utilize processors in its partition of participating machines.

A task distribution manager accumulates the load information of the nodes in the partition they control, and uses this information to choose the processor where a task should run.

In the Sprite system, every Sprite machine runs a background process called the "load-average daemon", which monitors the usage of the machine. When the machine appears idle, the daemon notifies the "central migration server" that the machine is prepared to accept migrated processes. User processes that invoke migration call a standard library routine, Mig_RequestIdleHosts, to obtain a list of idle hosts, and then reference the host identifier in the migrate process system call. The central migration server maintains the database in virtual memory, to avoid the overhead of remote filesystem operations. The load-average daemons and the library routine Mig_RequestIdleHosts communicate with the server using a message protocol. Sprite decides that a machine is idle if and only if (a) it had no keyboard or mouse inout for at least 30 seconds, and (b) there are, on average, fewer runnable processes than processors. This decision was made purely heuristically; originally the input threshold was 5 minutes. The Sprite designers chose not to determine the most efficient utilization of idle hosts, because there were plenty of idle hosts available. [7]

MOSIX is fully decentralized; every node acts as task distribution manager. At regular intervals, each node sends information about its available resources to a randomly chosen partition of nodes [9]. Each node therefore only maintains load information for a random partition of nodes, and will choose nodes among this set for the destination of the process migration. The use of randomness supports scaling and dynamic configuration [9].

3 Tradeoff Comparisons

The design of distributed operating systems involve making tradeoffs among four factors: transparency, residual dependencies, performance, and complexity. Perfect transparency would mean that both the user and the process act the same way to a remotely executing process as to a local one. Both the user and the process need not be aware of the fact that a process has been migrated. If remote execution leaves residual dependencies, that means the source machine must continue to provide services to the remotely executing process. By performance, we mean that that the remote execution mechanism should only induce minimal overheads in processing and allocation. The delay associated with initiating remote execution, or migrating a process, should be low, and remotely executing processes should perform as efficiently as locally executing ones. The complexity of the remote execution mechanism becomes important because it could potentially affect every piece of the operating system kernel. Depending on the relative importance of the remote execution mechanism to the designers of these operating systems, complexity may be limited for maintainability. [7]

These factors conflict with each other. High transparency are likely to be require more complexity and residual dependencies. Residual dependencies affect performance, because of the high delays associated with forwarding. A fast migration process may involve the use of residual dependencies to avoid the transfer of state, this can reduce the performance of the execution of the remote process. [7]

3.1 Sprite

The Sprite operating system guarantees transparency to remotely executing processes. The user can interact with a migrated process in the same manner as before migration took place. The user can continue to provide input to a process and receive output from it in an identical way. The user can also control the execution of the process using the same job control mechanisms provided for controlling local processes. No distinction is made between locally executing and migrated processes when using these job control mechanisms. However, Sprite requires the user-level application to initiate process migration. So for an application to take advantage of process migration, it not only must be aware of migration but it must determine when to request migration of subprocesses. Sprite does not automatically migrate processes except for eviction.

Sprite transfers most of the state associated with a process, but still retains some residual dependencies. Sprite transfers virtual memory, open file handles, and execution state. Access to file and memory are the most intensive operations, so elimination of residual dependencies in these areas tremendously improves performance. By restricting migration to the case of homogeneous processor architectures, the execution state transfer becomes simple. Forwarding is required for access to local I/O devices. For message channels between processes, the source machine must arrange to route messages for the migrated process. All signals are forwarded from the source machine. The state transfer and state forwarding mechanisms are implemented transparently. The only visible affect would be a reduction of performance when state forwarding is used instead of state transfer. In one case Sprite is not able to provide transparency, and that is for access to memory mapped I/O devices. Sprite simply forbids the migration of processes that use memory mapped I/O. [7]

3.2 MOSIX

MOSIX's progress migration mechanism is very similar to Sprite's. Both rely on a common file system to avoid the need to forward file operations. Virtual address space and execution state is transferred. However, unlike Sprite, only active pages are offloaded from the source machine. MOSIX doesn't store the backing store on the distributed file system, so the source system must be consulted when bringing in pages from the backing store. This is different from what occurs in Sprite. Sprite flushes all dirty pages back to the file server. The migrated process will page fault on every page it accesses, and load all pages from the file server. MOSIX's mechanism involves a residual dependency that Sprite does not, since the source machine must serve requests for virtual memory pages throughout the execution of the migrated process. However, in Sprite those pages would need to be demand loaded from the file server anyway. And MOSIX reduces the number of page faults during the initial stages of the migrated program's execution, because the active pages are transferred before execution begins. [9]

Unlike Sprite, the MOSIX kernel actively migrates processes using a load balancing algorithm rather than forcing the user-level application to make the request to migrate. This adds more transparency to the process migration mechanism. Not only are processes unaware of being migrated, the user also does not need to be aware of the need to migrate. This should also result in greater performance. Since the kernel gathers load information, it will automatically know when a processor becomes idle. In Sprite, user level programs can only ask which processors are idle at a given time; they cannot arrange to be notified when processors become idle.

3.3 Plan 9

Plan 9 supports transparency for remotely executing processes with the ability to export the local name space to the remote process. Since Plan 9 doesn't support process migration, state transfer is not required. Virtual memory is only allocated on the remote system. However, all references to files and devices are forwarded back to the local system using a network RPC protocol called 9P. The local system runs a program called exportfs, which translates 9P calls into system calls on the local machine. A forwarding mechanism is unavoidable for access to local devices. But other distributed systems do not forward accesses to files on distributed file systems to the source system, the dataserver can be referenced directly. Plan 9's mechanism sacrifices performance for the simplicity of residual dependencies. [6]

3.4 Clouds

The object-based paradigm of Clouds has many advantages over traditional systems. The view of the system to the user as a uniform, flat namespace of objects is conceptually simple. Object method invocation also provides the simplicity of procedural semantics.

The overhead associated with invocation, however, can incur a substantial performance penalty. There is no shared memory between objects. Input and output parameters are pass-by-value. In a thread which produces many invocations, the execution time can become dominated by the copying of parameters necessary for each invocation.

The nature in which a thread invokes a method, enters a object's virtual address space, and then continues execution in the invoked object has the benefit of not creating residual dependencies. The only information needed by the invokee is the set of input parameters from the originating object. [1]

3.5 Condor

Condor is a software system that runs on top of a UNIX kernel. This provides ease of portability and simplifies the task of operating system design. Complicated features of the operating system not directly related to the distributed aspect, such as device driver support, can be borrowed from the underlying operating system. This reduction in implementation complexity comes at the expense of system performance. Placing the distributed mechanisms outside the kernel incurs execution overhead and delay in passing load statistics and in load sharing decisions [11] .

When a job is submitted to a remote machine, a user is not required to have an account on the remote machine. The participating machines agree to allow other users to gain access and use the machines whenever the machines are idle. Since the users have no accounts on the remote machines, they cannot gain access to the remote machines' filesystem. However, when executing a process, it is possible that the process needs to read and write to a local file. Therefore, Condor needs to provide access to the local filesystem. Condor, then, needs to remotely execute system calls to the home machine when the running process needs access to the filesystem. This residual dependency on the home machine induces communication overhead for file operations.

3.6 Solaris MC

One of the primary goals of Solaris MC is to integrate distributed features into existing operating systems, namely Solaris, with maximum compatibility. The distributed structure of the system is transparant to the users and the applications. Byte level compatibilty is guaranteed for existing applications. Facilities to utilize remote CPU resources is provided. To minimize the increase of system complexity, modifications to the Solaris kernel is kept minimum. In exchange for achieving transparancy and minimum complexity, performance is sacrificed. For example, all file operation and system calls are interposed by the Solaris MC layer, performance of local file operation and local system calls will be lower. [4]

4 Further Improvements

4.1 Load Balancing Algorithms

An important goal of a distributed system goes beyond the ability to simply share resources. System designers are faced with doing this sharing efficiently and in a way that maximally utilizes resources. Response time and total throughput are the driving forces behind this work. The question then is how to balance the work load across all nodes in the system. Load balancing is also referred to as global scheduling.

Utilizing idle CPU resources via process migration has been discussed in an earlier section. With exception of MOSIX, the operating systems surveyed generally only provide the migration mechanism, and leave the policy implementation up to the application layer. The operating systems that provide this policy use algorithms that process load information gathered from each node to determine the destination when migrating a process. A standard load metric is the average number of tasks in the ready queue of a processor.

Gathering accurate load information from each node is the primary problem in load balancing. There are two basic models for storing this information: centralized and decentralized.

In the centralized model, a central server is used as storage of the load information, and therefore is given the responsibility for scheduling decisions. A fully decentralized system distributes this responsibility among the individual nodes.

Another issue in load balancing is the method of transmission of the load information. One method is to have nodes broadcast this information from time to time. Another method uses polling of the nodes for this information.

Polling to gather load information leads to a great number of messages being transmitted as requests and responses. The problem of high message traffic also occurs when having nodes broadcast their load information. This approach is not scalable. [2]

Lau et al, proposed a solution to the problem of messaging overhead for load information transmission in the decentralized scheduler model. This solution involves the use of anti-tasks and load state vectors. An anti-task is a special type of message that is passed among the computational nodes. The path of an anti-task is determined by the load state vector. An anti-task contains a table in which the entries are the load state values of the nodes that the anti-task has visited. Each of these entries is time-stamped and contains a visited flag. Each node has a table with the same structure, minus the visited flag. The table that the node maintains is called the load state vector, and the table on the anti-task side is called the anti-task's trajectory. When an anti-task visits a node, the information in each table is shared make sure that each table contains the most up to date information.

Using minimum and maximum threshold load values, a node is categorized as being in a light, normal, or heavy workload state. Lau et al, devised an algorithm that takes into account the information in the trajectory (load state information plus the visited flag) which cause anti-tasks to travel spontaneously towards the most heavily loaded nodes. The total information presented by arriving anti-tasks to the heavily loaded node give it a highly accurate view of the global state, increasing the chance that the node makes a good load balancing decision. [3]

4.2 Heterogeneous Process Migration

Marvin M. Theimer and Barry Hayes discuss an approach to migrating processes across heterogeneous processor architectures in their paper "Heterogeneous Process Migration By Recompilation". Since it is not possible to migrate the actual execution state of a process, which is machine dependant, the authors propose a method of constructing an equivalent machine independant state, which can be migrated. However, the approach can only work if the program is itself machine independant.

The technique requires that the compiler generate machine independant intermediate code along with the machine language code. The machine independant code will describe the operations on an abstract machine, while the machine language code describes operations for a physical machine. Compilers can be expected to optimize the machine code for a particular processor type, such that the internal states of the machine independant code and the machine language code will correspond only at a subset of execution points of a program. Such points are called migration points. When a migration is requested, the program will continue to execute until the next migration point. To keep delays in migration small, we'd like to have as many migration points as possible. If we allow migration delay in the range of seconds, there is room for millions of machine instructions to execute before we would need a migration point. It is necessary for all procedures in the call stack to have reached a state corresponding to an abstract state for the execution point to be considered a migration point.

Once we have reached such a migration point, we must generate an abstract program state. Compilers must generate source-level symbol tables describing the locations of every global and procedure-local variable, this is essentially what debugging features describe. We would use the same technique that a source-level debugger uses to gather the state of all global and procedure-local variables. Now that global and call stack data have been accounted for, we must find the state of the heap. The heap needs to be traced, following each pointer variable in the global and call stack to find the transitive closure of the objects they are pointing to. We must also be able to interpret every field of a heap object, because data representation conversions may be necessary across platforms (the size and representations of integers and floating point numbers often differ).

After accumulating this abstract state, we construct a "migration program" which initializes itself with the machine independant state and proceeds executes the rest of the code. This program is recompiled for the target system, and then migrated.

This approach can only be guaranteed to work, in the general case, for languages that themselves do not allow machine dependant code to be written. The authors believe their approach will work for Modula-2, type-safe Cedar, and Lisp. [8]

This paper preceded the development of Java. Java is designed such that source code is converted into a machine independant byte code. This byte code runs on top of a Java virtual machine process. Remote execution of Java bytecode will of course require no recompilation, but migrating a Java process requires transferring the state of the virtual machine. The Java virtual machine satisfies exactly the requirements of the abstract machine described by the authors. It is no longer necessary to determine migration points, every execution point in a Java bytecode is a migration point. The same mechanism for locating and recording the values of all global, procedure-local, and heap data can be applied to the Java Virtual Machine.

5 Conclusion

In this paper, we've analyzed distributed operating systems that offered widely varied approaches. On one end of the spectrum, we have Clouds: an operating system designed from the ground up to be distributed. Its programming model is drastically different from standard methodologies used today, because the programming model has been reinvented to support the idea of disjoint distributed resources. Clouds requires a total rethinking of program design, but at the same time provides the most simple and efficient distributed computing model. Clouds avoids the complexities and overhead associated with transferring of state, Clouds only transfers procedure arguments across distributed objects. If distributed computing is the primary feature a programmer is seeking, then the opportunity cost of adapting the programs from the traditional and familiar model could be justified. However, a general purpose programmer who doesn't depend on distributed computing abilities would see Cloud's peculiarities as a nuisance.

On the other end of the spectrum, we have distributed systems like Solaris MC and Condor. Both of these provide distributed computing through a user-level layer that sits above a UNIX operating system. Very minimal, if any, changes to the kernel need be made. The layer provides transparent distributed task management by using existing features of the operating system. The result is a large amount of overhead and significant reduction in performance. However, the cost of maintaining the distributed operating system is simplified. These layered systems can rely on the vender of the general purpose operating system to maintain the most volatile components of operating systems, such as supporting new hardware devices. Since the underlying operating system caters to a more general user base, the issue of providing operating system support for the small community of distributed computer system users is greatly alleviated.

However, these layer based distributed systems do have severe performance issues. It would be convenient if there could be a way to implement these distributed system features within a popular kernel and still be able to manage this code separately from the rest of the kernel. Sprite attempted to provide an efficient process migration mechanism in their kernel, but chose not to automate it. The main reason for this was that there were many different goals of the Sprite project, only one of which was distributed task management. The Sprite designers wanted to minimize the effect process migration would have on other developing parts of the operating system kernel. So they prevented the kernel from actively migrating processes, so that they could allow developers to test parts of the operating system independantly of the effects of process migration.

MOSIX attempts to provide distributing system features by designing kernel extensions to popular operating systems, such as BSD/OS and Linux. The MOSIX developers produce source code patches for particular versions of the Linux kernel, and thereby allowing MOSIX to be built as a kernel module. The efficiency of kernel mode distributed task management support is coupled with the advantage of integration with a widely used and maintained mainstream operating system.

In conclusion, we predict that the approach that MOSIX takes is most likely to be successful in integrating distributed task management features into mainstream operating systems.

References

[1] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, Umakishore Ramachandran, *"The Clouds Distributed Operating System,"* IEEE Computer, Volume 24, 1991.

[2] Marvin M. Theimer, Keith A. Lantz, *"Finding Idle Machines in a Workstation-based Distributed System,"* IEEE Trans. on Parallel and Distributed Systems, 1988.

[3] Sau-Ming Lau, Qin Lu, Kwong-Sak Leung, *"Dynamic Load Distribution Using Anti-Tasks and Load State Vectors,"* IEEE Trans. on Parallel and Distributed Systems, 1988.

[4] Yousef A. Khalidi, Jose Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadani, *"Solaris MC: A multicomputer OS,"* Proceedings of 1996 USENIX Conference, January 1996.

[5] Ken Shirriff, *"Building Distributed Process Management on an Object-Oriented Framework* USENIX 1997

[6] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom, *"Plan 9 from Bell Labs"*,1995

[7] John K. Ousterhout, Frederick Douglis, *"Transparent Process Migration: Design Alternatives and the Sprite Implementation"* Software—Practice & Experience, August 1991.

[8] Marvin M. Theimer, Barry Hayes, *"Heterogeneous Process Migration by Recompilation",* IEEE 11th Int'l Conference on Distributed Computing Systems, 1991

[9] Amnon Barak, Oren La'adan, *"The MOSIX Multicomputer Operating System for High Performance Cluster Computing,"* 1997.

[10] K.G. Shin and C.-J. Hou, *"Design and evaluation of effective load sharing in distributed real-time systems,"* IEEE Trans. on Parallel and Distributed Systems, vol. 5, no.,7, July 1994.

[11] Chao-Ju Hou, Kang G. Shin, *"Implementation of Decentralized Load Sharing in Networked Workstations Using the Condor Package,"* 1994