Reducing Downtime Due to System Maintenance and Upgrades

Shaya Potter and Jason Nieh – Columbia University

ABSTRACT

Patching, upgrading, and maintaining operating system software is a growing management complexity problem that can result in unacceptable system downtime. We introduce AutoPod, a system that enables unscheduled operating system updates while preserving application service availability. AutoPod provides a group of processes and associated users with an isolated machine-independent virtualized environment that is decoupled from the underlying operating system instance. This virtualized environment is integrated with a novel checkpoint-restart mechanism which allows processes to be suspended, resumed, and migrated across operating system kernel versions with different security and maintenance patches.

AutoPod incorporates a system status service to determine when operating system patches need to be applied to the current host, then automatically migrates application services to another host to preserve their availability while the current host is updated and rebooted. We have implemented AutoPod on Linux without requiring any application or operating system kernel changes. Our measurements on real world desktop and server applications demonstrate that AutoPod imposes little overhead and provides sub-second suspend and resume times that can be an order of magnitude faster than starting applications after a system reboot. AutoPod enables systems to autonomically stay updated with relevant maintenance and security patches, while ensuring no loss of data and minimizing service disruption.

Introduction

As computers become more ubiquitous in large corporate, government, and academic organizations, the total cost of owning and maintaining them is becoming unmanageable. Computers are increasingly networked, which only complicates the management problem, given the myriad of viruses and other attacks commonplace in today's networks. Security problems can wreak havoc on an organization's computing infrastructure. To prevent this, software vendors frequently release patches that can be applied to address security and maintenance issues that have been discovered. This creates a management nightmare for administrators who take care of large sets of machines. For these patches to be effective, they need to be applied to the machines. It is not uncommon for systems to continue running unpatched software long after a security exploit has become well-known [22]. This is especially true of the growing number of server appliances intended for very low-maintenance operation by less skilled users. Furthermore, by reverse engineering security patches, exploits are being released as soon as a month after the fix is released, whereas just a couple of years ago, such exploits took closer to a year to create [12].

Even when software updates are applied to address security and maintenance issues, they commonly result in system services being unavailable. Patching an operating system can result in the entire system having to be down for some period of time. If a system administrator chooses to fix an operating system security problem immediately, he risks upsetting his users because of loss of data. Therefore, a system administrator must schedule downtime in advance and in cooperation with users, leaving the computer vulnerable until repaired. If the operating system is patched successfully, the system downtime may be limited to just a few minutes during the reboot. Even then, users are forced to incur additional inconvenience and delays in starting applications again and attempting to restore their sessions to the state they were in before being shutdown. If the patch is not successful, downtime can extend for many hours while the problem is diagnosed and a solution is found. Downtime due to security and maintenance problems is not only inconvenient but costly as well.

We present AutoPod, a system that provides an easy-to-use autonomic infrastructure [11] for operating system self-maintenance. AutoPod uniquely enables unscheduled operating system updates of commodity operating systems while preserving application service availability during system maintenance. AutoPod provides its functionality without modifying, recompiling, or relinking applications or operating system kernels. This is accomplished by combining three key mechanisms: a lightweight virtual machine isolation abstraction that can be used at the granularity of individual applications, a checkpoint-restart mechanism that operates across operating system versions with different security and maintenance patches, and an autonomic system status service that monitors the system for system faults as well as security updates.

AutoPod provides a lightweight virtual machine abstraction called a POD (PrOcess Domain) that encapsulates a group of processes and associated users in an isolated machine-independent virtualized environment that is decoupled from the underlying operating system instance. A pod mirrors the underlying operating system environment but isolates processes from the system by using host-independent virtual identifiers for operating system resources. Pod isolation not only protects the underlying system from compromised applications, but is crucial for enabling applications to migrate across operating system instances. Unlike hardware virtualization approaches that require running multiple operating system instances [4, 29, 30], pods provide virtual application execution environments within a single operating system instance. By operating within a single operating system instance, pods can support finer granularity isolation and can be administered using standard operating system utilities without sacrificing system manageability. Furthermore, since it does not run an operating system instance, a pod prevents potentially malicious code from making use of an entire set of operating system resources.

AutoPod combines its pod virtualization with a novel checkpoint-restart mechanism that uniquely decouples processes from dependencies on the underlying system and maintains process state semantics to enable processes to be migrated across different machines. The checkpoint-restart mechanism introduces a platform-independent intermediate format for saving the state associated with processes and AutoPod virtualization. AutoPod combines this format with the use of higher-level functions for saving and restoring process state to provide a high degree of portability for process migration across different operating system versions that was not possible with previous approaches. In particular, the checkpoint-restart mechanism relies on the same kind of operating system semantics that ensure that applications can function correctly across operating system versions with different security and maintenance patches.

AutoPod combines the pod virtual machine with an autonomous system status service. The service monitors the system for system faults as well as security updates. When the service detects new security updates, it is able to download and install them automatically. If the update requires a reboot, the service uses the pod's checkpoint-restart capability to save the pod's state, reboot the machine into the newly fixed environment, and restart the processes within the pod without causing any data loss. This provides fast recovery from system downtime even when other machines are not available to run application services. Alternatively, if another machine is available, the pod can be migrated to the new machine while the original machine is maintained and rebooted, further minimizing application service downtime. This enables security patches to be applied to operating systems in a timely manner with minimal impact on the availability of application services. Once the original machine has been updated, applications can be returned and can continue to execute even though the underlying operating system has changed. Similarly, if the service detects an imminent system fault, AutoPod can checkpoint the processes, migrate, and restart them on a new machine before the fault can cause the processes' execution to fail.

We have implemented AutoPod in a prototype system as a loadable Linux kernel module. We have used this prototype to securely isolate and migrate a wide range of unmodified legacy and network applications. We measure the performance and demonstrate the utility of AutoPod across multiple systems running different Linux 2.4 kernel versions using three real-world application scenarios, including a full KDE desktop environment with a suite of desktop applications, an Apache/MySQL web server and database server environment, and a Exim/Procmail e-mail processing environment. Our performance results show that AutoPod can provide secure isolation and migration functionality on real world applications with low overhead.

This paper describes how AutoPod can enable operating system self-maintenance by suspending, resuming, and migrating applications across operating system kernel changes to facilitate kernel maintenance and security updates with minimal application downtime. Subsequent sections describe the AutoPod virtualization abstractions, present the virtualization architecture to support the AutoPod model, discuss the AutoPod checkpoint-restart mechanisms used to facilitate migration across operating system kernels that may differ in maintenance and security updates, provide a brief overview of the AutoPod system status service, provide a security analysis of the AutoPod system as well as examples of how to use AutoPod, and present experimental results evaluating the overhead associated with AutoPod virtualization and quantifying the performance benefits of AutoPod migration versus a traditional maintenance approach for several application scenarios. We discuss related work before some concluding remarks.

AutoPod Model

The AutoPod model is based on a virtual machine abstraction called a pod. Pods were previously introduced in Zap [16] to support migration assuming the same operating system version is used for all systems. AutoPod extends this work to enable pods to provide a complete secure virtual machine abstraction in addition to heterogeneous migration functionality. A pod looks just like a regular machine and provides the same application interface as the underlying operating system. Pods can be used to run any application, privileged or otherwise, without modifying, recompiling, or relinking applications. This is essential for both ease-of-use and protection of the underlying system, since applications not executing in a pod offer an opportunity to attack the system. Processes within a pod can make use of all available operating system services, just like processes executing in a traditional operating system environment. Unlike a traditional operating system, the pod abstraction provides a self-contained unit that can be isolated from the system, checkpointed to secondary storage, migrated to another machine, and transparently restarted.

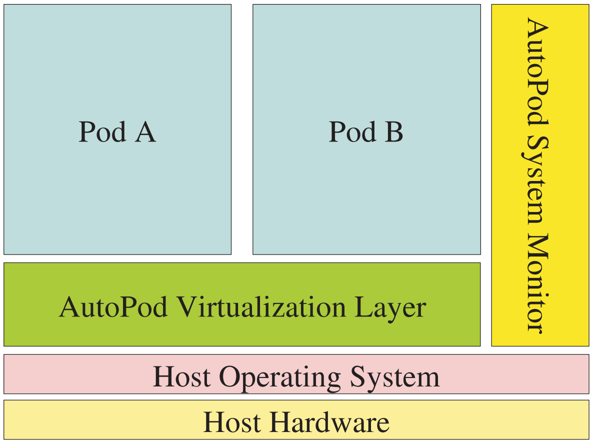

Figure 1: The AutoPod model.

AutoPod enables server consolidation by allowing multiple pods to be in use on a single machine, while enabling automatic machine status monitoring as shown in Figure 1. Since each pod provides a complete secure virtual machine abstraction, they are able to run any server application that would run on a regular machine. By consolidating multiple machines into distinct pods running on a single server, one improves manageability by limiting the number of physical hardware and the number of operating system instances an administrator has to manage. Similarly, when kernel security holes are discovered, server consolidation improves manageability by minimizing the amount of machines that need to be upgraded and rebooted. The AutoPod system monitor further improves manageability by constantly monitoring the host system for stability and security problems.

Since a pod does not run an operating system instance, it provides a virtualized machine environment by providing a host-independent virtualized view of the underlying host operating system. This is done by providing each pod with its own virtual private namespace. All operating system resources are only accessible to processes within a pod through the pod's virtual private namespace.

A pod namespace is private in that only processes within the pod can see the namespace. It is private in that it masks out resources that are not contained within the pod. Processes inside a pod appear to one another as normal processes that can communicate using traditional Inter-Process Communication (IPC) mechanisms. Other processes outside a pod do not appear in the namespace and are therefore not able to interact with processes inside a pod using IPC mechanisms such as shared memory or signals. Instead, processes outside the pod can only interact with processes inside the pod using network communication and shared files that are normally used to support process communication across machines.

A pod namespace is virtual in that all operating system resources including processes, user information, files, and devices are accessed through virtual identifiers within a pod. These virtual identifiers are distinct from host-dependent resource identifiers used by the operating system. The pod virtual namespace provides a host-independent view of the system by using virtual identifiers that remain consistent throughout the life of a process in the pod, regardless of whether the pod moves from one system to another. Since the pod namespace is distinct from the host's operating system namespace, the pod namespace can preserve this naming consistency for its processes even if the underlying operating system namespace changes, as may be the case in migrating processes from one machine to another. This consistency is essential to support process migration [16].

The pod private, virtual namespace enables secure isolation of applications by providing complete mediation to operating system resources. Pods can restrict what operating system resources are accessible within a pod by simply not providing identifiers to such resources within its namespace. A pod only needs to provide access to resources that are needed for running those processes within the pod. It does not need to provide access to all resources to support a complete operating system environment. An administrator can configure a pod in the same way one configures and installs applications on a regular machine. Pods enforce secure isolation to prevent exploited pods from being used to attack the underlying host or other pods on the system. Similarly, the secure isolation allows one to run multiple pods from different organizations, with different sets of users and administrators on a single host, while retaining the semantic of multiple distinct and individually managed machines.

For example, to provide a web server, one can easily setup a web server pod to only contain the files the web server needs to run and the content it wants to serve. The web server pod could have its own IP address, decoupling its network presence from the underlying system. The pod can have its network access limited to client-initiated connections using firewall software to restrict connections to the pod's IP address to only the ports served by the application running within this pod. If the web server application is compromised, the pod limits the ability of an attacker to further harm the system since the only resources he has access to are the ones explicitly needed by the service. The attacker cannot use the pod to directly initiate connections to other systems to attack them since the pod is limited to client-initiated connections. Furthermore, there is no need to carefully disable other network services commonly enabled by the operating system to protect against the compromised pod since those services, and the core operating system itself, reside outside of the pod's context.

AutoPod Virtualization

To support the AutoPod abstraction design of secure and isolated namespaces on commodity operating systems, we employ a virtualization architecture that operates between applications and the operating system, without requiring any changes to applications or the operating system kernel. This virtualization layer is used to translate between the AutoPod namespaces and the underlying host operating system namespace. It protects the host operating system from dangerous privileged operations that might be performed by processes within the AutoPod, as well as protecting those processes from processes outside of the AutoPod.

Pods are supported using virtualization mechanisms that translate between the pod's resource identifiers and the operating system's resource identifiers. Every resource that a process in a pod accesses is through a *virtual private name* which corresponds to an operating system resource identified by a *physical name*. When an operating system resource is created for a process in a pod, such as with process or IPC key creation, instead of returning the corresponding physical name to the process, the pod virtualization layer catches the physical name value, and returns a virtual private name to the process. Similarly, any time a process passes a virtual private name to the operating system, the virtualization layer catches it and replaces it with the appropriate physical name.

The key pod virtualization mechanisms used are a system call interposition mechanism and the chroot utility with file system stacking to provide each pod with its own file system namespace that can be separate from the regular host file system. Pod virtualization support for migration is based on Zap [16]. We focus here on pod virtualization support for secure virtual machine isolation.

Because current commodity operating systems are not built to support multiple namespaces, AutoPod must take care of the security issues this causes. While chroot can give a set of processes a virtualized file system namespace, there are many ways to break out of the standard chrooted environment, especially if one allows the chroot system call to be used by processes in a pod. Pod file system virtualization enforces the chrooted environment and ensures that the pod's file system is only accessible to processes within the given pod by using a simple form of file system stacking to implement a barrier. File systems provide a permission function that determines if a process can access a file.

For example, if a process tries to access a file a few directories below the current directory, the permission function is called on each directory as well as the file itself in order. If any of the calls determine that the process does not have permission on a directory, the chain of calls end. Even if the permission function would determine that the process would have access to the file itself, it must have permission to traverse the directory hierarchy to the file to access it.

We implement a barrier by simply stacking a small pod-aware file system on top of the staging directory that overloads the underlying permission function to prevent processes running within the pod from accessing the parent directory of the staging directory, and to prevent processes running only on the host from accessing the staging directory. This effectively confines a process in a pod to the pod's file system by preventing it from ever walking past the pod's file system root.

While any network file system can be used with pods to support migration, we focus on NFS because it is the most commonly used network file system. Pods can take advantage of the user identifier (UID) security model in NFS to support multiple security domains on the same system running on the same operating system kernel. For example, since each pod can have its own private file system, each pod can have its own /etc/passwd file that determines its list of users and their corresponding UIDs. In NFS, the UID of a process determines what permissions it has in accessing a file.

By default, pod virtualization keeps process UIDs consistent across migration and keeps process UIDs the same in the pod and operating system namespaces. However, since the pod file system is separate from the host file system, a process running in the pod is effectively running in a separate security domain from another process with the same UID that is running directly on the host system. Although both processes have the same UID, each process is only allowed to access files in its own file system namespace. Similarly, multiple pods can have processes running on the same system with the same UID, but each pod effectively provides a separate security domain since the pod file systems are separate from one another.

The pod UID model supports an easy-to-use migration model when a user may be using a pod on a host in one administrative domain and then moves the pod to another. Even if the user has computer accounts in both administrative domains, it is unlikely that the user will have the same UID in both domains if they are administratively separate. Nevertheless, pods can enable the user to run the same pod with access to the same files in both domains.

Suppose the user has UID 100 on a machine in administrative domain A and starts a pod connecting to a file server residing in domain A. Suppose that all pod processes are then running with UID 100. When the user moves to a machine in administrative domain B where he has UID 200, he can migrate his pod to the new machine and continue running processes in the pod. Those processes can continue to run as UID 100 and continue to access the same set of files on the pod file server, even though the user's real UID has changed. This works, even if there's a regular user on the new machine with a UID of 100. While this example considers the case of having a pod with all processes running with the same UID, it is easy to see that the pod model supports pods that may have running processes with many different UIDs.

Because the root UID 0 is privileged and treated specially by the operating system kernel, pod virtualization treats UID 0 processes inside of a pod specially as well. AutoPod is required to do this to prevent processes running with privilege from breaking the pod abstraction, accessing resources outside of the pod, and causing harm to the host system. While a pod can be configured for administrative reasons to allow full privileged access to the underlying system, we focus on the case of pods for running application services which do not need to be used in this manner. Pods do not disallow UID 0 processes, which would limit the range of application services that could be run inside pods. Instead, pods provide restrictions on such processes to ensure that they function correctly inside of pods.

While a process is running in user space, its UID does not have any affect on process execution. Its UID only matters when it tries to access the underlying kernel via one of the kernel entry points, namely devices and system calls. Since a pod already provides a virtual file system that includes a virtual /dev with a limited set of secure devices, the device entry point is already secured. The only system calls of concern are those that could allow a root process to break the pod abstraction. Only a small number of system calls can be used for this purpose. These system calls are listed and described in further detail in the Appendix. Pod virtualization classifies these system calls into three classes.

The first class of system calls are those that only affect the host system and serve no purpose within a pod. Examples of these system calls include those that load and unload kernel modules or that reboot the host system. Since these system calls only affect the host, they would break the pod security abstraction by allowing processes within it to make system administrative changes to the host. System calls that are part of this class are therefore made inaccessible by default to processes running within a pod.

The second class of system calls are those that are forced to run unprivileged. Just like NFS, by default, squashes root on a client machine to act as user nobody, pod virtualization forces privileged processes to act as the nobody user when they want to make use of some system calls. Examples of these system calls include those that set resource limits and ioctl system calls. Since system calls such as setrlimit and nice can allow a privileged process to increase its resource limits beyond predefined limits imposed on pod processes, privileged processes are by default treated as unprivileged when executing these system calls within a pod. Similarly, the ioctl system call is a system call multiplexer that allows any driver on the host to effectively install its own set of system calls. Since the ability to audit the large set of possible system calls is impossible given that pods may be deployed on a wide range of machine configurations that are not controlled by the AutoPod system, pod virtualization conservatively treats access to this system call as unprivileged by default.

The final class of system calls are calls that are required for regular applications to run, but have options that will give the processes access to underlying host resources, breaking the pod abstraction. Since these system calls are required by applications, the pod checks all their options to ensure that they are limited to resources that the pod has access to, making sure they are not used in a manner that breaks the pod abstraction. For example, the mknod system call can be used by privileged processes to make named pipes or files in certain application services. It is therefore desirable to make it available for use within a pod. However, it can also be used to create device nodes that provide access to the underlying host resources. To limit how the system call is used, the pod system call interposition mechanism checks the options of the system call and only allows it to continue if it is not trying to create a device.

Migration Across Different Kernels

To maintain application service availability without losing important computational state as a result of system downtime due to operating system upgrades, AutoPod provide a checkpoint-restart mechanism that allows pods to be migrated across machines running different operating system kernels. Upon completion of the upgrade process, the respective AutoPod and its applications are restored on the original machine. We assume here that any kernel security holes on the unpatched system have not yet been exploited on the system; migrating across kernels that have already been compromised is beyond the scope of this paper. We also limit our focus to migrating between machines with a common CPU architecture with kernel differences that are limited to maintenance and security patches. These patches often correspond to changes in the minor version number of the kernel. For example, the Linux 2.4 kernel has nearly thirty minor versions. Even within minor version changes, there can be significant changes in kernel code. Table 1 shows the number of files that have been changed in various subsystems of the Linux 2.4 kernel across different minor versions. For example, all of the files for the VM subsystem were changed since extensive modifications were made to implement a completely new page replacement mechanism in Linux.

Many of the Linux kernel patches contain security vulnerability fixes, which are typically not separated out from other maintenance patches. We similarly limit our focus to where the application's execution semantics, such as how threads are implemented and how dynamic linking is done, do not change. On the Linux kernels this is not an issue as all these semantics are enforced by user-space libraries. Whether one uses kernel or user threads, or how libraries are dynamically linked into a process is all determined by the respective libraries on the file system. Since the pod has access to the same file system on whatever machine it is running on, these semantics stay the same.

To support migration across different kernels, AutoPod use a checkpoint-restart mechanism that employs an intermediate format to represent the state that needs to be saved on checkpoint. On checkpoint, the intermediate format representation is saved and digitally signed to enable the restart process to verify the integrity of the image. Although the internal state that the kernel maintains on behalf of processes can be different across different kernels, the high-level properties of the process are much less likely to change. We capture the state of a process in terms of higher-level semantic information specified in the intermediate format rather than kernel specific data in native format to keep the format portable across different kernels.

For example, the state associated with a UNIX socket connection consists of the directory entry of the UNIX socket file, its superblock information, a hash key, and so on. It may be possible to save all of this state in this form and successfully restore on a different machine running the same kernel. But this representation of a UNIX socket connection state is of limited portability across different kernels. A different high-level representation consisting of a four tuple, virtual source PID, source FD, virtual destination PID, destination FD is highly portable. This is because the semantics of a process identifier and a file descriptor are typically standard across different kernels, especially across minor version differences.

The intermediate representation format used by AutoPod for migration is chosen such that it offers the degree of portability needed for migrating between different kernel minor versions. If the representation of state is too high-level, the checkpoint-restart mechanism could become complicated and impose additional overhead. For example, the AutoPod system saves the address space of a process in terms of discrete memory regions called virtual memory (VM) areas. As an alternative, it may be possible to save the contents of a process's address space and denote the characteristics of various portions of it in more abstract terms. However, this would call for an unnecessarily complicated interpretation scheme and make the implementation inefficient. The VM area abstraction is standard across major Linux kernel revisions. AutoPod view the VM area abstraction as offering sufficient portability in part because the organization of a process's address space in this manner has been standard across all Linux kernels and has never changed.

AutoPod further support migration across different kernels by leveraging higher-level native kernel services to transform intermediate representation of the checkpointed image into an internal representation suitable for the target kernel. Continuing with the previous example, AutoPod restore a UNIX socket connection using high-level kernel functions as follows. First, two new processes are created with virtual PIDs as specified in the four tuple. Then, each one creates a UNIX socket with the specified file descriptor and one socket is made to connect to the other. This procedure effectively recreates the original UNIX socket connection without depending on many kernel internal details.

This use of high-level functions helps in general portability of using AutoPod for migration. Security patches and minor version kernel revisions commonly involve modifying the internal details of the kernel while high-level primitives remain unchanged. As such services are usually made available to kernel modules, the AutoPod system is able to perform cross-kernel migration without requiring modifications to the kernel code.

The AutoPod checkpoint-restart mechanism is also structured in such a way to perform its operations

Type	2.4.1	2.4.29	Modified	Unmodified	% Unmodified
Drivers	2623	3784	1742	501	13.2
Arch	123	128	93	22	17.1
FS	536	894	410	59	6.6
Network	461	600	338	84	9.4
Core Kernel	27	27	24	3	11.1
VM	21	20	20	0	0
IPC	6	6	5	1	16.6

Table 1: Kernel file changes within the Linux 2.4 series for i386.

when processes are in a state that checkpointing can avoid depending on many low-level kernel details. For example, semaphores typically have two kinds of state associated with each of them: the value of the semaphore and the wait queue of processes waiting to acquire the corresponding semaphore lock. In general, both of these pieces of information have to be saved and restored to accurately reconstruct the semaphore state. Semaphore values can be easily obtained and restored through GETALL and SETALL parameters of the semctl system call. But saving and restoring the wait queues involves manipulating kernel internals directly. The AutoPod mechanism avoids having to save the wait queue information by requiring that all the processes be stopped before taking the checkpoint. When a process waiting on a semaphore receives a stop signal, the kernel immediately releases the process from the wait queue and returns EINTR. This ensures that the semaphore wait queues are always empty at the time of checkpoint so that they do not have to be saved.

While AutoPod can abstract and manipulate most process state in higher-level terms using higher-level kernel services, there are some parts that not amenable to a portable intermediate representation. For instance, specific TCP connection states like timestamp values and sequence numbers, which do not have a high-level semantic value, have to be saved and restored to maintain a TCP connection. As this internal representation can change, its state needs to be tracked across kernel versions and security patches. Fortunately, there is usually an easy way to interpret such changes across different kernels because networking standards such as TCP do not change often. Across all of the Linux 2.4 kernels, there was only one change in TCP state that required even a small modification in the AutoPod migration mechanism. Specifically, in the Linux 2.4.14 kernel, an extra field was added to TCP connection state to address a flaw in the existing syncookie mechanism. If configured into the kernel, syncookies protect an Internet server against a synflood attack. When migrating from an earlier kernel to a Linux-2.4.14 or later version kernel, the AutoPod system initializes the extra field in such a way that the integrity of the connection is maintained. In fact, this was the only instance across all of the Linux 2.4 kernel versions where an intermediate representation was not possible and the internal state had changed and had to be accounted for.

To provide proper support for AutoPod virtualization when migrating across different kernels, we must ensure that that any changes in the system call interfaces are properly accounted for. As AutoPod has a virtualization layer using system call interposition mechanism for maintaining namespace consistency, a change in the semantics for any system call intercepted by AutoPod could be an issue in migrating across different kernel versions. But such changes usually do not occur as it would require that the libraries be rewritten. In other words, AutoPod virtualization is protected from such changes in a similar way as legacy applications are protected. However, new system calls could be added from time to time. For instance, across all Linux 2.4 kernels to date, there were two new system calls, gettid and tkill for querying the thread identifier and for sending a signal to a particularly thread in a thread group, respectively, which needed to be accounted for to properly virtualize AutoPod across kernel versions. As these system calls take identifier arguments, they were simply intercepted and virtualized.

Autonomic System Status Service

AutoPod provides a generic autonomic framework for managing system state. The framework is able to monitor multiple sources for information and can use this information to make autonomic decisions about when to checkpoint pods, migrate them to other machines, and restart them. While there are many items that can be monitored, our service monitors two items in particular. First, it monitors the vendor's software security update repository to ensure that the system stays up to date with the latest security patches. Second, it monitors the underlying hardware of the system to ensure that an imminent fault is detected before the fault occurs and corrupts application state. By monitoring these two sets of information, the autonomic system status service is able to reboot or shutdown the computer, while checkpointing or migrating the processes. This helps ensure that data is not lost or corrupted due to a forced reboot or a hardware fault propagating into the running processes.

Many operating system vendors provide their users with the ability to automatically check for system updates and to download and install them when they become available. Example of these include Microsoft's Windows Update service, as well as Debian based distribution's security repositories. Users are guaranteed that the updates one gets through these services are genuine because they are verified through cryptographic signed hashes that verify the contents as coming from the vendors. The problem with these updates is that some of them require machine reboots; In the case of Debian GNU/Linux this is limited to kernel upgrades. We provide a simple service that monitors these security update repositories. The autonomic service simply downloads all security updates, and by using the pod's checkpoint/restart mechanism enables the security updates that need reboots to take effect without disrupting running applications and causing them to lose state.

Commodity systems also provide information about the current state of the system that can indicate if the system has an imminent failure on its hands. Subsystems, such as a hard disk's Self-Monitoring Analysis Reporting Technology (SMART), let an autonomic service monitor the system's hardware state. SMART provides diagnostic information, such as temperature and read/write error rates, on the hard drives in the system that can indicate if the hard disk is nearing failure. Many commodity computer motherboards also have the ability to measure CPU and case temperature, as well as the speeds of the fans that regulate those temperatures. If temperature in the machine rises too high, hardware in the machine can fail catastrophically. Similarly, if the fans fail and stop spinning, the temperature will likely rise out of control. Our autonomic service monitors these sensors and if it detects an imminent failure, will attempt to migrate an AutoPod to a cooler system, as well as shutdown the machine to prevent the hardware from being destroyed.

Many administrators use an uninterruptible power supply to avoid having a computer lose or corrupt data in the event of a power loss. While one can shutdown a computer when the battery backup runs low, most applications are not written to save their data in the presence of a forced shutdown. AutoPod, on the other hand, monitors UPS status and if the battery backup becomes low can quickly checkpoint the pod's state to avoid any data loss when the computer is forced to shutdown.

Similarly, the operating system kernel on the machine monitors the state of the system, and if irregular conditions occur, such as DMA timeout or needing to reset the IDE bus, will log this occurrence. Our autonomic service monitors the kernel logs to discover these irregular conditions. When the hardware monitoring systems or the kernel logs provide information about possible pending system failures, the autonomic service checkpoints the pods running on the system, and migrates them to a new system to be restarted on. This ensures state is not lost, while informing system administrators the a machine needs maintenance.

Many policies can be implemented to determine which system a pod should be migrated to while a machine needs maintenance. Our autonomic service uses a simple policy of allowing a pod to be migrated around a specified set of clustered machines. The autonomic service gets reports at regular intervals from the other machines' autonomic services that reports each machine's load. If the autonomic service decides that it must migrate a pod, it chooses the machine in its cluster that has the lightest load.

Security Analysis

Saltzer and Schroeder [24] describe several principles for designing and building secure systems. These include:

- **Economy of mechanism**: Simpler and smaller systems are easier to understand and ensure that they do not allow unwanted access.
- **Complete mediation**: Systems should check every access to protected objects.
- **Least privilege**: A process should only have access to the privileges and resources it needs to do its job.
- **Psychological acceptability**: If users are not willing to accept the requirements that the security system imposes, such as very complex passwords that the users are forced to write down, security is impaired. Similarly, if using the system is too complicated, users will misconfigure it and end up leaving it wide open.
- **Work factor**: Security designs should force an attacker to have to do extra work to break the system. The classic quantifiable example is when one adds a single bit to an encryption key, one doubles the key space an attacker has to search.

AutoPod is designed to satisfy these five principles. AutoPod provides economy of mechanism using a thin virtualization layer based on system call interposition and file system stacking that only adds a modest amount of code to a running system. Furthermore, AutoPod changes neither applications nor the underlying operating system kernel. The modest amount of code to implement AutoPod makes the system easier to understand. Since the AutoPod security model only provides resources that are physically within the environment, it is relatively easy to understand the security properties of resource access provided by the model.

AutoPod provides for complete mediation of all resources available on the host machine by ensuring that all resources accesses occur through the pod's virtual namespace. Unless a file, process, or other operating system resource was explicitly placed in the pod by the administrator or created within the pod, AutoPod's virtualization will not allow a process within a pod to access the resource.

AutoPod provides a least privilege environment by enabling an administrator to only include the data necessary for each service. AutoPod can provide separate pods for individual services so that separate services are isolated and restricted to the appropriate set of resources. Even if a service is exploited, AutoPod will limit the attacker to the resources the administrator provided for that service. While one can achieve similar isolation by running each individual service on a separate machine, this leads to inefficient use of resources. AutoPod maintains the same least privilege semantic of running individual services on separate machines, while making efficient use of machine resources at hand. For instance, an administrator could run MySQL and Exim mail transfer services on a single machine, but within different pods. If the Exim pod gets exploited, the pod model ensures that the MySQL pod and its data will remain isolated from the attacker.

AutoPod provides psychological acceptability by leveraging the knowledge and skills system administrators already use to setup system environments. Because pods provide a virtual machine model, administrators can use their existing knowledge and skills to run their services within pods. This differs from other least privilege architectures that force an administrator to learn new principles or complicated configuration languages that require a detailed understanding of operating system principles.

AutoPod increases the work factor required to compromise a system by not making available the resources that attackers depend on to harm a system once they have broken in. For example, services like mail delivery do not depend on having access to a shell. By not including a shell program within a mail delivery AutoPod, one makes it difficult for an attacker to get a root shell that they would use to further their attacks. Similarly, the fact that one can migrate a system away from a host that is vulnerable to attack increases the work an attacker would have to do to make services unavailable.

AutoPod Examples

We briefly describe two examples that help illustrate how AutoPod can be used to improve application availability for different application scenarios. The application scenarios are system services, such as e-mail delivery and desktop computing. In both cases we describe the architecture of the system and show how it can be run within AutoPod, enabling administrators to reduce downtime in the face of machine maintenance. We also discuss how a system administrator can setup and use pods.

System Services

Administrators like to run many services on a single machine. By doing this, they are able to benefit from improved machine utilization, but at the same time give each service access to many resources they do not need to perform their job. A classic example of this is e-mail delivery. E-mail delivery services, such as Exim, are often run on the same system as other Internet services to improve resource utilization and simplify system administration through server consolidation. However, services such as Exim have been easily exploited by the fact that they have access to system resources, such as a shell program, that they do not need to perform their job.

For e-mail delivery, AutoPod can isolate e-mail delivery to provide a significantly higher level of security in light of the many attacks on mail transfer agent vulnerabilities that have occurred. Consider isolating an Exim service, the default Debian mail transfer agent, installation. Using AutoPod, Exim can execute in a resource restricted pod, which isolates e-mail delivery from other services on the system. Since pods allow one to migrate a service between machines, the e-mail delivery pod is migratable. If a fault is discovered in the underlying host machine, the e-mail delivery service can be moved to another system while the original host is patched, preserving the availability of the e-mail service.

With this e-mail delivery example, a simple system configuration can prevent the common buffer overflow exploit of getting the privileged server to execute a local shell. This is done by just removing shells from within the Exim pod, thereby limiting the amateur attacker's ability to exploit flaws while requiring very little additional knowledge about how to configure the service. AutoPod can further automatically monitor system status and checkpoint the Exim pod if a fault is detected to ensure that no data is lost or corrupted. Similarly, in the event that a machine has to be rebooted, the service can automatically be migrated to a new machine to avoid any service downtime.

A common maintenance problem system administrators face is that forced machine downtime, for example due to reboots, can cause a service to be unavailable for a period of time. A common way to avoid this problem is to throw multiple machines at the problem. By providing the service through a cluster of machines, system administrators can upgrade the individual machines in a rolling manner. This enables system administrators to upgrade the systems providing the service while keeping the service available. The problem with this solution is that system administrators need to throw more machines at the problem than they might need to provide the service effectively, thereby increasing management complexity as well as cost.

AutoPod in conjunction with hardware virtual machine monitors improves this situation immensely. Using a virtual machine monitor to provide two virtual machines on a single host, AutoPod can then run a pod within a virtual machine to enable a single node maintenance scenario that can decrease costs as well management complexity. During regular operation, all application services run within the pod on one virtual machine. When one has to upgrade the operating system in the running virtual machine, one brings the second virtual machine online and migrates the pod to the new virtual machine.

Once the initial virtual machine is upgraded and rebooted, the pod can be migrated back to it. This reduces costs as only a single physical machine is needed. This also reduces management complexity as only one virtual machine is in use for the majority of the time the service is in operation. Since AutoPod runs unmodified applications, any application service that can be installed can make use of AutoPod's ability to provide general single node maintenance.

Desktop Computing

As personal computers have become more ubiquitous in large corporate, government, and academic organizations, the total cost of owning and maintaining them is becoming unmanageable. These computers are increasingly networked which only complicates the management problem. They need to be constantly patched and upgraded to protect them, and their data, from the myriad of viruses and other attacks commonplace in today's networks.

To solve this problem, many organizations have turned to thin-client solutions such as Microsoft's Windows Terminal Services and Sun's Sun Ray. Thin clients give administrators the ability to centralize many of their administrative duties as only a single computer or a cluster of computers needs to be maintained in a central location, while stateless client devices are used to access users' desktop computing environments. While thin-client solutions provide some benefits for lowering administrative costs, this comes at the loss of semantics users normally expect from a private desktop. For instance, users who use their own private desktop expect to be isolated from their coworkers. However, in a shared thin-client environment, users share the same machine. There may be many shared files and a user's computing behavior can impact the performance of other users on the system.

While a thin-client environment minimizes the machines one has to administrate, the centralized servers still need to be administrated, and since they are more highly utilized, management becomes more difficult. For instance, on a private system one only has to schedule system maintenance with a single user, as reboots will force the termination of all programs running on the system. However, in a thin-client environment, one has to schedule maintenance with all the users on the system to avoid having them lose any important data.

AutoPod enables system administrators to solve these problems by allowing each user to run a desktop session within a pod. Instead of users directly sharing a single file system, AutoPod provides each pod with a composite of three file systems: a shared read-only file system of all the regular system files users expect in their desktop environments, a private writable file system for a user's persistent data, and a private writable file system for a user's temporary data. By sharing common system files, AutoPod provides centralization benefits that simplify system administration. By providing private writable file systems for each pod, AutoPod provides each user with privacy benefits similar to a private machine.

Coupling AutoPod virtualization and isolation mechanisms with a migration mechanism can provide scalable computing resources for the desktop and improve desktop availability. If a user needs access to more computing resources, for instance while doing complex mathematical computations, AutoPod can migrate that user's session to a more powerful machine. If maintenance needs to be done on a host machine, AutoPod can migrate the desktop sessions to other machines without scheduling downtime and without forcefully terminating any programs users are running.

Setting Up and Using AutoPod

To demonstrate how simple it is to setup a pod to run within the AutoPod environment, we provide a step by step walkthrough on how one would create a new pod that can run the Exim mail transfer agent. Setting up AutoPod to provide the Exim pod on Linux is straightforward and leverages the same skill set and experience system administrators already have on standard Linux systems. AutoPod is started by loading its kernel module into a Linux system and using its user-level utilities to setup and insert processes into a pod.

Creating a pod's file system is the same as creating a chroot environment. Administrators that have experience creating a minimal environment, that just contains the application they want to isolate, do not need to do any extra work. However, many administrators do not have experience creating such an environment and therefore need an easy way to create an environment to run their application in. These administrators can take advantage of Debian's debootstrap utility that enables a user to quickly setup an environment that's the equivalent of a base Debian installation. An administrator would do a `debootstrap stable /pod` to install the most recently released Debian system into the `/pod` directory. While this will also include many packages that are not required by the installation, it provides a small base to work from. An administrator can remove packages, such as the installed mail transfer agent, that are not needed.

To configure Exim, an administrator edits the appropriate configuration files within the `/pod/etc/exim4/` directory. To run Exim in a pod, an administrator does `mount -o bind /pod /autopod/exim/root` to loopback mount the pod directory onto the staging area directory where AutoPod expects it. `autopod add exim` is used to create a new pod named exim which uses `/autopod/exim/root` as the root for its file system. Finally, `autopod addproc exim /usr/sbin/exim4` is used to start Exim within the pod by executing the `/usr/sbin/exim4` program, which is actually located at `/autopod/exim/root/usr/sbin/exim4`.

AutoPod isolates the processes running within a pod from the rest of the system, which helps contain intrusions if they occur. However, since a pod does not have to be maintained by itself, but can be maintained in the context of a larger system, one can also prune down the environment and remove many programs that an attacker could use against the system. For instance, if an Exim pod has no need to run any shell scripts, there is no reason an administrator has to leave programs such as `/bin/bash`, `/bin/sh` and `/bin/dash` within the environment. One issue is that these programs are necessary if the administrator wants to be able to simply upgrade the package in the future via normal Debian methods. Since it is simple to recreate the environment, one approach would be to remove all the programs that are not wanted within the environment and recreate the environment when an upgrade is needed. Another approach would be to move those programs outside of the pod, such as by creating a `/pod-backup` directory. To upgrade the pod using the normal Debian package upgrade methods, the programs can then be moved back into the pod file system.

If an administrator wants to manually reboot the system without killing the processes within this Exim pod, the administrator can first checkpoint the pod to disk by running autopod checkpoint exim -o /exim.pod, which tells AutoPod to checkpoint the processes associated with the exim pod to the file /exim.pod. The system can then be rebooted, potentially with an updated kernel. Once it comes back up, the pod can be restarted from the /exim.pod file by running autopod restart exim -i /exim.pod. These mechanisms are the same as those used by the AutoPod system status service for controlling the checkpointing and migration of pods.

Standard Debian facilities for installing packages can be used for running other services within a pod. Once the base environment is setup, an administrator can chroot into this environment by running chroot /pod to continue setting it up. By editing the /etc/apt/sources.list file appropriately and running apt-get update, an administrator will be able to install any Debian package into the pod. In the Exim example, Exim does not need to be installed since it is the default MTA and already included in the base Debian installation. If one wanted to install another MTA, such as Sendmail, one could run apt-get install sendmail, which will download Sendmail and all the packages needed to run it. This will work for any service available within Debian. An administrator can also use the dpkg --purge option to remove packages that are not required by a given pod. For instance, in running an Apache web server in a pod, one could remove the default Exim mail transfer agent since it is not needed by Apache.

Experimental Results

We implemented AutoPod as a loadable kernel module in Linux, that requires no changes to the Linux kernel, as well as a user space system status monitoring service. We present some experimental results using our Linux prototype to quantify the overhead of using AutoPod on various applications. Experiments were conducted on a trio of IBM Netfinity 4500R machines, each with a 933 Mhz Intel Pentium-III CPU, 512 MB RAM, 9.1 GB SCSI HD and a 100 Mbps Ethernet connected to a 3Com Superstack II 3900 switch. One of the machines was used as an NFS server from which directories were mounted to construct the virtual file system for the AutoPod on the other client systems. The clients ran different Linux distributions and kernels, one machine running Debian Stable with a Linux 2.4.5 kernel and the other running Debian Unstable with a Linux 2.4.18 kernel.

To measure the cost of AutoPod virtualization, we used a range of micro benchmarks and real application workloads and measured their performance on our Linux AutoPod prototype and a vanilla Linux system. Table 2 shows the seven micro-benchmarks and four application benchmarks we used to quantify AutoPod virtualization overhead as well as the results for a vanilla Linux system. To obtain accurate measurements, we rebooted the system between measurements. Additionally, the system call micro-benchmarks directly used the TSC register available on Pentium CPUs to record timestamps at the significant measurement events. Each timestamp's average cost was 58 ns. The files for the benchmarks were stored on the NFS Server. All of these benchmarks were performed in a chrooted environment on the NFS client machine running Debian Unstable with a Linux 2.4.18 kernel. Figure 4 shows the results of running the benchmarks under both configurations, with the vanilla Linux configuration normalized to one. Since all benchmarks measure the time to run the benchmark, a small number is better for all benchmarks results.

The results in Figure 2 show that AutoPod virtualization overhead is small. AutoPod incurs less than 10% overhead for most of the micro-benchmarks and less than 4% overhead for the application workloads. The overhead for the simple system call getpid benchmark is only 7% compared to vanilla Linux, reflecting the fact that AutoPod virtualization for these kinds of system calls only requires an extra procedure call and

Name	Description	Linux
getpid	average getpid runtime	350 ns
ioctl	average runtime for the FIONREAD ioctl	427 ns
shmget-shmctl	IPC Shared memory segment holding an integer is created and removed	3361 ns
semget-semctl	IPC Semaphore variable is created and removed	1370 ns
fork-exit	process forks and waits for child which calls exit immediately	44.7 us
fork-sh	process forks and waits for child to run /bin/sh to run a program that prints "hello world" then exits	3.89 ms
Apache	Runs Apache under load and measures average request time	1.2 ms
Make	Linux Kernel compile with up to 10 process active at one time	224.5 s
Postmark	Use Postmark Benchmark to simulate Exim performance	.002 s
MySQL	"TPC-W like" interactions benchmark	8.33 s

Table 2: Application benchmarks.

a hash table lookup. The most expensive benchmarks for AutoPod is semget+semctl which took 51% longer than vanilla Linux. The cost reflects the fact that our untuned AutoPod prototype needs to allocate memory and do a number of namespace translations.

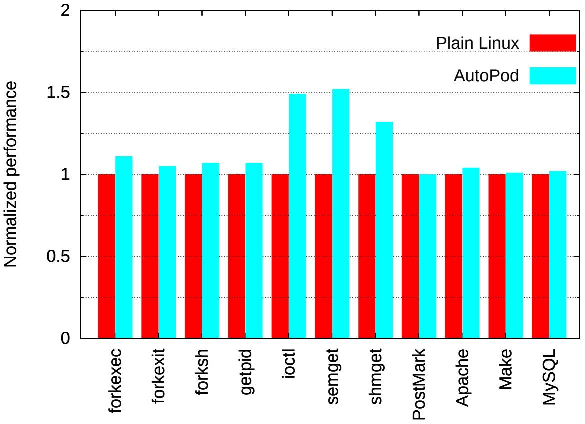

Figure 2: AutoPod virtualization overhead.

The ioctl benchmark also has high overhead, because of the 12 separate assignments it does to protect the call against malicious root processes. This is large compared to the simple FIONREAD ioctl that just performs a simple dereference. However, since the ioctl is simple, we see that it only adds 200 ns of overhead over any ioctl. For real applications, the most overhead was only four percent which was for the Apache workload, where we used the http_load benchmark [18] to place a parallel fetch load on the server with 30 clients fetching at the same time. Similarly, we tested MySQL as part of a web-commerce scenario outlined by TPC-W with a bookstore servlet running on top of Tomcat with a MySQL back-end. The AutoPod overhead for this scenario was less than 2% versus vanilla Linux.

To measure the cost of AutoPod migration and demonstrate the ability of AutoPod to migrate real applications, we migrated the three application scenarios; an email delivery service using Exim and Procmail, a web content delivery service using Apache and MySQL, and a KDE desktop computing environment. Table 3 described the configurations of the application scenarios we migrated, as well as showing the time it takes to startup on a regular Linux system. To demonstrate our AutoPod prototype's ability to migrate across Linux kernels with different minor versions, we checkpointed each application workload on the 2.4.5 kernel client machine and restarted it on the 2.4.18 kernel machine. For these experiments, the workloads were checkpointed to and restarted from local disk.

Case	Check Point	Restart	Size	Compr'd
E-mail	11 ms	14 ms	284 KB	84 KB
Web	308 ms	47 ms	5.3 MB	332 KB
Desktop	851 ms	942 ms	35 MB	8.8 MB

Table 4: AutoPod migration costs.

Table 4 shows the time it took to checkpoint and restart each application workload. In addition to these, migration time also has to take into account network transfer time. As this is dependent on the transport medium, we include the uncompressed and compressed checkpoint image sizes. In all cases, checkpoint and restart times were significantly faster than the regular startup times listed in Table 5, taking less than a second for both operations, even when performed on separate machines or across a reboot. We also show that the actual checkpoint images that were saved were modest in size for complex workloads. For example, the Desktop pod had over 30 different processes running, providing the KDE desktop applications, as well as substantial underlying window system infrastructure, including inter-application sharing, a rich desktop interface managed by a window manager with a number of applications running in a panel such as the clock. Even with all these applications running, they checkpoint to a very reasonable 35 MB uncompressed for a full desktop

Name	Applications	Normal Startup
E-mail	Exim 3.36	504 ms
Web	Apache 1.3.26 and MySQL 4.0.14 .	2.1 s
Desktop	Xvnc – VNC 3.3.3r2 X Server KDE – Entire KDE 2.2.2 environment, including window manager, panel and assorted background daemon and utilities SSH – openssh 3.4p1 client inside a KDE konsole terminal connected to a remote host Shell – The Bash 2.05a shell running in a konsole terminal KGhostView – A PDF viewer with a 450 KB 16 page PDF file loaded. Konqueror – A modern standards compliant web browser that is part of KDE KOffice – The KDE word processor and spreadsheet programs	19 s

Table 3: Application scenarios.

environment. Additionally, if one needed to transfer the checkpoint images over a slow link, Table 6 shows that they can be compressed very well with the bzip2 compression program.

Related Work

Virtual machine monitors (VMMs) have been used to provide secure isolation [4, 29, 30], and have also been used to to migrate an entire operating system environment [25]. Unlike AutoPod, standard VMMs decouple processes from the underlying machine hardware, but tie them to an instance of an operating system. As a result, VMMs cannot migrate processes apart from that operating system instance and cannot continue running those processes if the operating system instance ever goes down, such as during security upgrades. In contrast, AutoPod decouples process execution from the underlying operating system which allows it to migrate processes to another system when an operating system instance is upgraded. VMMs have been proposed to support online maintenance of systems [14] by having a microvisor that supports at most two virtual machines running on the machine at the same time, effectively giving each physical machine the ability to act as its own hot spare. However, this proposal explicitly depends on AutoPod migration functionality yet does not provide it.

A number of other approaches have explored the idea of virtualizing the operating system environment to provide application isolation. FreeBSD's Jail mode [10] provides a chroot like environment that processes can not break out of. However, since Jail is limited in what it can do, such as the fact it does not allow IPC within a jail [9] many real world application will not work. More recently, Linux Vserver [1] and Solaris Zones [19] offer a similar virtual machine abstraction as AutoPod pods, but require substantial in-kernel modifications to support the abstraction. They do not provide isolation of migrating applications across independent machines, and have no support for maintaining application availability in the presence of operating system maintenance and security upgrades.

Many systems have been proposed to support process migration [2, 3, 6, 7, 8, 13, 15, 17, 20, 21, 23, 26], but do not allow migration across independent machines running different operating system versions. TUI [27] provides support for process migration across machines running different operating systems and hardware architectures. Unlike AutoPod, TUI has to compile applications on each platform using a special compiler and does not work with unmodified legacy applications. AutoPod builds on a pod abstraction introduced in Zap [16] to support transparent migration across systems running the same kernel version. Zap does not address security issues or heterogeneous migration. AutoPod goes beyond Zap in providing a complete, secure virtual machine abstraction for isolating processes, finer-grain mechanisms for isolating application components, and transparent migration across minor kernel versions, which is essential for providing application availability in the presence of operating system security upgrades.

Replication in clustered systems can provide the ability to do rolling upgrades. By leveraging many nodes, individual nodes can be taken down for maintenance, without significantly impacting the load the cluster can handle. For example, web content is commonly delivered by multiple web servers behind a front end manager. This front end manager enables an administrator to bring down back end web servers for maintenance as it will only direct requests to the active web servers. This simple solution is effective because it is easy to replicate web servers to serve the same content. While this model works fine for web server loads, as the individual jobs are very short, it does not work for long running jobs, such as a user's desktop. In the web server case, replication and upgrades are easy to do since only one web server is used to serve any individual request and any web server can be used to serve any request. For long running stateful applications, such as a user's desktop, requests cannot be arbitrarily redirected to any desktop computing environment as each user's desktop session is unique. While specialized hardware support could be used to keep replicas synchronized, by having all of them process all operations, this is prohibitively expensive for most workloads and does not address the problem of how to resynchronize the replicas in the presence of rolling upgrades.

Another possible solution to this problem is allowing the kernel to be hot plugable. While micro-kernels are not prevalent, they contain this ability to upgrade their parts on the fly. More commonly, many modern monolithic kernels have kernel modules that can be inserted and removed dynamically. This can allow one to upgrade parts of a monolithic kernel without requiring any reboots. The Nooks [28] system extends this concept by enabling kernel drivers and other kernel functionality, such as file systems, to be isolated into their own protection domain to help isolate faults in kernel code and provide a more reliable system. However, in all of these cases, their is still a base kernel on the machine that cannot be replaced without a reboot. If one has to replace that part, all data would be lost.

The K42 operating system has the ability to be dynamically updated [5]. This functionality enables software patches to be applied to a running kernel even in the presence of data structure changes. However, it requires a completely new operating system design and does not work with any commodity operating system. Even on K42, it is not yet possible to upgrade the kernel while running realistic application workloads.

Conclusions

The AutoPod system provides an operating system virtualization layer that decouples process execution from the underlying operating system, by running the process within a pod. Pods provide an easy-to-use lightweight virtual machine abstraction that can securely isolate individual applications without the need to run an operating system instance in the pod. Furthermore, AutoPod can be transparently migrate isolated applications across machines running different operating system kernel versions. This enables security patches to be applied to operating systems in a timely manner with minimal impact on the availability of application services. It also preserves secure isolation of untrusted applications in the presence of migration.

We have implemented AutoPod on Linux without requiring any application or operating system kernel changes. We demonstrated how pods can be used to enable autonomic machine maintenance and increase availability for a range of applications, including e-mail delivery, web servers with databases and desktop computing. Our measurements on real world applications demonstrate that AutoPod imposes little overhead, provides sub-second suspend and resume times that can be an order of magnitude faster than starting applications after a system reboot, and enables systems to autonomously stay updated with relevant maintenance and security patches, while ensuring no loss of data and minimizing service disruption.

Acknowledgments

Matt Selsky contributed to the architecture of the isolation mechanism. Dinesh Subhraveti contributed to the implementation of the process migration mechanism. This work was supported in part by NSF grants CNS-0426623 and ANI-0240525, and an IBM SUR Award.

Author Information

Shaya Potter is a Ph.D. candidate in Columbia University's Computer Science department. His research interests are focused around improving computer usage for users and administrators through virtualization and process migration technologies. He received his B.A. from Yeshiva University and his M.S. and M.Phil degrees from Columbia University, all in Computer Science. Reach him electronically at spotter@cs.columbia.edu .

Jason Nieh is an Associate Professor of Computer Science at Columbia University and Director of the Network Computing Laboratory. He is also the technical adviser for nine States on the Microsoft Antitrust Settlement. He received his B.S. from MIT and his M.S. and Ph.D. from Stanford University, all in Electrical Engineering. Reach him electronically at nieh@cs.columbia.edu .

Bibliography

[1] *Linux VServer Project,* http://www.linux-vserver.org/ .

[2] Artsy, Y, Y. Chang, and R. Finkel, "Interprocess communication in charlotte," *IEEE Software*, pages 22-28, January, 1987.

[3] Barak, A. and R. Wheeler, "MOSIX: An Integrated Multiprocessor UNIX," *Proceedings of the USENIX Winter 1989 Technical Conference*, pp. 101-112, San Diego, CA, February, 1989.

[4] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October, 2003.

[5] Baumann, A., J. Appavoo, D. D. Silva, J. Kerr, O. Krieger, and R. W. Wisniewski, "Providing dynamic update in an operating system," *USENIX 2004*, pp. 279-291, Anaheim, California, April, 2005.

[6] Casas, J., D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "MPVM: A migration transparent version of PVM," *Computing Systems*, Vol. 8, Num. 2, pp. 171-216, 1995.

[7] Cheriton, D., "The V distributed system," *Communications of the ACM*, Vol. 31, Num. 3, pp. 314-333, March, 1988.

[8] Douglis, F. and J. Ousterhout, "Transparent process migration: Design alternatives and the sprite implementation," *Software – Practice and Experience*, Vol. 21, Num. 8, pp. 757-785, August, 1991.

[9] FreeBSD Project, *Developer's handbook,* http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/secure-chroot.html .

[10] Kamp, P.-H. and R. N. M. Watson, "Jails: Confining the omnipotent root," *2nd International SANE Conference*, MECC, Maastricht, The Netherlands, May, 2000.

[11] Kephart, J. O., and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, pages 41-50, January, 2003.

[12] LaMacchia, B., Personal Communication, January, 2004.

[13] Litzkow, M., T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and migration of unix processes in the condor distributed processing system*, Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April, 1997.

[14] Lowell, D. E., Y. Saito, and E. J. Samberg, "Devirtualizable virtual machines enabling general, single-node, online maintenance," *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October, 2004.

[15] Mullender, S. J., G. v. Rossum, A. S. Tanenbaum, R. v. Renesse, and H. v. Staveren, "Amoeba: a distributed operating system for the 1990s," *IEEE Computer*, Vol. 23, Num. 5, pp. 44-53, May, 1990.

[16] Osman, S. D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for

Migrating Computing Environments," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December, 2002.

[17] Plank, J. S., M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under UNIX," *Proceedings of Usenix Winter 1995 Technical Conference*, pp. 213-223, New Orleans, LA, January, 1995.

[18] Poskanzer, J., http://www.acme.com/software/http_load/ .

[19] Price, D. and A. Tucker, "Solaris zones: Operating system support for consolidating commercial workloads," *18th Large Installation System Administration Conference (LISA 2004)*, November, 2004.

[20] Pruyne, J. and M. Livny, "Managing checkpoints for parallel programs," *2nd Workshop on Job Scheduling Strategies for Parallel Processing (In Conjunction with IPPS '96)*, Honolulu, Hawaii, April, 1996.

[21] Rashid R. and G. Robertson, "Accent: A communication oriented network operating system kernel," *Proceedings of the 8th Symposium on Operating System Principles*, pp. 64-75, December, 1984.

[22] Rescorla, E., "Security holes... Who cares?" *Proceedings of the 12th USENIX Security Conference*, Washington, D. C., August, 2003.

[23] Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "Overview of the Chorus distributed operating system," *Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 39-70, Seattle, WA, 1992.

[24] Saltzer, J. H., and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, Vol. 63, Num. 9, pp. 1278-1308, September, 1975.

[25] Sapuntzakis, C. P., R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[26] Schmidt, B. K., *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*, Ph.D. thesis, Computer Science Department, Stanford University, 2000.

[27] Smith, P. and N. C. Hutchinson, "Heterogeneous process migration: The Tui system," *Software – Practice and Experience*, Vol. 28, Num. 6, pp. 611-639, 1998.

[28] Swift, M. M., B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 207-222, ACM Press, New York, NY, 2003.

[29] VMware, Inc., http://www.vmware.com .

[30] Whitaker, A., M. Shaw, and S. D. Gribble, "Scale and Performance in the Denali Isolation Kernel," *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December, 2002.

Appendix

To isolate regular Linux processes within a pod, AutoPod interposes on a number of system calls. Below we provide a complete list of the small number of system calls that require more than plain virtualization. We give the reasoning for the interposition and what functionality was changed from the base system call. Most system calls do not require more than simple virtualization to ensure isolation because virtualization of the resources itself takes care of the isolation. For example, the kill system call can not signal a processes outside of a pod because the virtual private namespace will not map them and therefore it cannot reference it.

Host Only System Calls

1. mount – If a user within a regular pod is able to mount a file system, they could mount a file system with device nodes already present and thus would be able to access the underlying system directly in a manner that is not controlled by AutoPod. Therefore, regular pod processes are prevented from using this system call.
2. stime, adjtimex – These system call enable a privileged process to adjust the host's clock. If a user within a regular pod could call this system call they would cause a change on the host. Therefore regular pod processes are prevented from using this system call.
3. acct – This system call sets what file on the host BSD process accounting information should be written to. As this is host specific functionality, AutoPod prevents regular pod processes from using this system call.
4. swapon, swapoff – These system calls control swap space allocation. Since these system calls are host specific and have no use within a regular pod, AutoPod prevents regular pod processes from calling these system calls.
5. reboot – This system call can cause the system to reboot or change Ctrl-Alt-Delete functionality and therefore serves no place inside a regular pod. AutoPod therefore disallows regular pod processes from calling it.
6. ioperm, iopl – These system calls enable a privileged process to gain direct access to underlying hardware resources. Since regular pod processes do not access hardware directly, AutoPod prevents regular pod process from calling these system calls.

7. create_module, init_module, delete_module, query_module – These system calls are only related to inserting and removing kernel modules. As this is a host specific function, AutoPod prevents regular pod processes from calling these system calls.
8. sethostname, setdomainname – These system call sets the name for the underlying host. AutoPod wraps these system calls to save it as a pod specific name and allows each pod to call it independently.
9. nfsservctl – This system call can enable a privileged process inside a pod to change the host's internal NFS server. AutoPod therefore prevents a process within a regular pods from calling this system call.

Root Squashed System Calls

1. nice, setpriority, sched_setscheduler – These system calls lets a process change its priority. If a process is running as root (UID 0), it can increase its priority and freeze out other processes on the system. Therefore, AutoPod prevents any regular pod process from increasing its priority.
2. ioctl – This system call is a syscall demultiplexer that enables kernel device drivers and subsystems to add their own functions that can be called from user space. However, as functionality can be exposed that enables root to access the underlying host, all system call beyond a limited audited safe set are squashed to user nobody, similar to what NFS does.
3. setrlimit – this system call enables processes running as uid 0 to raise their resource limits beyond what was preset, thereby enabling them to disrupt other processes on the system by using too much resources. AutoPod therefore prevents regular pod processes from using this system call to increase the resources available to them.
4. mlock, mlockall – These system calls enable a privileged process to pin an arbitrary amount of memory, thereby enabling a pod process to lock all of available memory and starve all the other processes on the host. AutoPod therefore squashes a privileged processes to user nobody when it attempts to call this system call to treat it like a regular process.

Option Checked System Calls

1. mknod – This system call enables a privileged user to make special files, such pipes, sockets and devices as well as regular files. Since a privileged process needs to make use of such functionality, the system call cannot be disabled. However, if the process could create a device it be creating an access point to the underlying host system. Therefore when a regular pod process makes use of this system call, the options are checked to prevent it from creating a device special file, while allowing the other types through unimpeded.