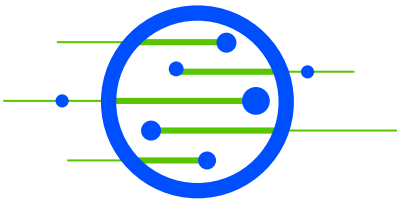

Proper: Privileged Operations in a Virtualised System Environment

Steve Muir, Larry Peterson, Marc Fiuczynski
Princeton University

Justin Cappos, John Hartman
University of Arizona

PDN–04–022

August 2004 (updated June 2005)

Appears in the *Proceedings of the USENIX 2005 Annual Technical Conference*, Anaheim, California, April 2005

Proper: Privileged Operations in a Virtualised System Environment

Steve Muir, Larry Peterson, Marc Fiuczynski, Justin Cappos, John Hartman
Princeton University and the University of Arizona
{smuir,llp,mef}@cs.princeton.edu, {justin,jhh}@cs.arizona.edu

Abstract

Virtualised systems have experienced a resurgence in popularity in recent years, whether used to support multiple OSes running on a user's desktop, provide commercial application hosting facilities, or isolate a large number of users from each other in global network testbeds. We also see an increasing level of interest in having entities within these virtualised systems interact with each other, either as peers or as helpers providing a service to clients.

Very little work has been previously conducted on how such interaction between virtualised environments can take place. We introduce Proper, a service running on the PlanetLab system, that allows unprivileged entities to access privileged operations in a safe, tightly controlled manner.

This paper describes our work designing and implementing Proper, including a discussion of the various architectural decisions made. We describe how implementing such a system in a traditional UNIX environment is non-trivial, and provide a number of examples of how services running on PlanetLab actually use Proper.

1 Introduction

Operating systems face a fundamental tension between providing isolation and sharing among applications—they simultaneously support the illusion that each application has the physical machine to itself, yet allow applications to share objects (e.g., files, pipes) with each other. OSes designed for personal computers (adapted from earlier time-sharing systems) typically provide a relatively weak form of isolation (the process abstraction) with generous facilities for sharing (e.g., a global file system and global process ids). In contrast, virtual machine monitors (VMMs) strive to provide strong performance isolation and privacy between virtual machines (VMs), and provide no more support for sharing between VMs than the network provides between physical machines.

The point on the design spectrum supported by any given system depends on the workload it is designed to support. Desktop OSes generally run multiple applications on behalf of a single user, making it natural to favor sharing over isolation. Similarly, VMMs are often designed to allow a single machine to host multiple independent applications, possibly running on behalf of independent organizations, as might be the case in a hosting center. In such a scenario, the applications have no need to share information and it is important they receive a predictable fraction of the physical machine's resources, hence, VMMs heavily favor isolation over sharing.

This paper investigates an alternative design point, motivated by the desire to allow VMs to interact with each other in well-defined and controlled ways. While all systems must provide a means by which isolated components interact, we are particularly interested in the problem of unbundling the management of a set of VMs from the underlying VMM. To enable multiple *management services*, it is necessary to 'poke holes' in the isolation barriers between VMs in a controlled manner. These holes allow one VM to see and manipulate objects such as files and processes in another VM, providing a natural means for one VM to help manage another. We refer to these 'holes' as *Controlled Virtual Machine Escapes*.

Toward this end, this paper describes Proper, a new PRivileged OPERations mechanism that VMs can use to poke the holes that they need. Proper provides global capabilities that can be passed between VMs on a single node and is implemented entirely using Linux system calls. Proper is straightforward to implement on a UNIX-based VMM such as that used in PlanetLab, and enables useful management services that run today on PlanetLab.

2 Virtualised Environment

This section investigates the isolation/sharing issue in depth, starting with a general discussion of isolation. Our goal is not to present a comprehensive taxonomy of isolation, but rather to tease apart the requirements that motivate Proper.

Our discussion considers the full breadth of system-level support for virtual machines, from traditional OSes to low-level hypervisors. For clarity, we settle on a single set of terminology, drawn from the recent virtual machine literature. That is, we refer to the underlying system as a virtual machine monitor (VMM) rather than an OS, and

the isolated containers running on top of it as virtual machines rather than processes or domains.

2.1 Isolation Landscape

Isolation between VMs involves two largely independent dimensions: *performance isolation* and *namespace isolation*. Note that we do not use the term *fault isolation* in our characterization of VMMs, instead taking the view that fault isolation is implied by performance isolation: a VMM that provides performance isolation protects all correctly running VMs from both greedy and faulty VMs. After all, a fault that crashes the machine robs all VMs of the performance they expect.

Performance isolation corresponds to the VMM's ability to isolate the resource consumption of one VM from that of another VM; undesired interactions between VMs is sometimes called cross-talk [12]. Providing performance isolation generally involves careful scheduling and allocation of physical machine resources (e.g., cycles, link bandwidth, disk space), but can also be influenced by VMs sharing logical resources, such as file descriptors and memory buffers. A VMM that supports strong performance isolation might guarantee that a VM will receive 100 million cycles per second (Mcps) and 1.5Mbps of link bandwidth, independent of any other applications running on the machine. On the other hand, a VMM that supports weak performance isolation might allow VMs to compete with each other for cycles and bandwidth on a demand-driven (best-effort) basis. Many hybrid approaches are possible; for instance, a VMM may maintain strong performance isolation between classes of VMs while enforcing weaker isolation within each class (e.g., to support a low priority class of applications that are allowed to consume excess cycles).

Namespace isolation refers to the extent to which the VMM limits access to (and information about) logical objects, such as files, memory addresses, port numbers, user IDs, process IDs, and so on. A VMM that supports strong namespace isolation does not reveal the names of files or process IDs belonging to another VM, let alone allow one VM to access or manipulate such objects. In contrast, a VMM that supports weak namespace isolation might support a shared namespace (e.g., a global file system), augmented with an access control mechanism that limits the ability of one VM to manipulate the objects owned by another VM.

The level of namespace isolation supported by a VMM affects at least two aspects of application programs. The first is *configuration independence*, that is, whether names (e.g., of files) selected by one VM possibly conflict with names selected by another VM. The second aspect is security; if one VM is not able to see or modify data and code belonging to another VM, then this increases the likelihood that a compromise to one VM does not affect others on the same machine.

Taken together, these two dimensions of isolation provide a simple map of the design space, with different VMMs (OSes) corresponding to each possibility:

- Traditional timesharing systems offer both weak performance isolation and weak namespace isolation.
- Single-user multimedia systems offer strong performance isolation but weak namespace isolation.
- A traditional timesharing system augmented with support for security contexts, as exemplified by Linux Vservers [13] and BSD jails [10], offer weak performance isolation and strong namespace isolation.
- A VMM designed for application hosting centers typically offer both strong performance isolation and strong namespace isolation.

Keep in mind that this section puts forward just one possible view of the isolation landscape. There are certainly other criteria by which VMM strategies can be evaluated—e.g., ease of porting guest OSes to the VMM, size of the trusted code base, scalability—but they are orthogonal to the issue of how to trade isolation against sharing.

2.2 New Requirement

We are interested in scenarios that require both performance and namespace isolation, but with the additional requirement that the VMM provide a means for the hosted VMs to directly interact with each other in well-defined and controlled ways. This might happen, for example, on a single-user machine that hosts untrusted third-party software, as well as in planetary-scale (grid-like) settings where services distributed across a network of hosting sites want to interact with each other. In other words, we are interested in scenarios in which each service runs in its own VM, but we want to support arbitrary service composition.

Two distinct approaches exist to the problem of giving one VM the ability to access another. The first is for the VMM to maintain strict namespace isolation and VMs to interact with each other over the network. The possibility that the two VMs are on the same physical machine is ignored. The alternative is to selectively circumvent namespace isolation to allow one VM to directly access another VM on the same machine, that is, to create well-defined *holes* in the VM isolation barrier by granting one VM privileges to access state in another VM. This paper describes just such a mechanism.

While there are potentially many scenarios in which a pair of VMs might want to interact, one of the more compelling occurs when one VM is responsible for *managing* or *controlling* the other. The idea is that rather than consolidate all management functionality in a single privileged VM (root domain) on a given machine, VM management should itself be implemented inside one or more

unprivileged VMs; a principle called *unbundled management* [3]. This allows the management capabilities of the system to evolve independent of the underlying VMM, and for the hosted applications to pick and chose from a larger set of management services (i.e., to utilize third-party software). In essence, unbundling the management of a VMs potentially creates a market for management services.

In this scenario, if a management service running in one VM is forced to control a client service running in another VM via the network, the former must acquire a vantage point inside the latter, from which the manager can manipulate the contents of the client. The management service could supply the client VM with a daemon to run, and then connect to this daemon through the network. The alternative, once again, is to selectively define holes in the namespace isolation that allows the management VM to directly control the client VM. Of course, this only works when the two services run on the same machine, but this is the nature of management services.

Clearly, there are alternatives to allowing management VMs to "poke holes". One is to consolidate all privileges to manipulate other VMs in a single root VM, as happens in Xen [2], for example. The problem with this approach is that it limits the ability of multiple management services to evolve at the same time. It also means that all management services are granted full root privileges, whereas our strategy gives management VMs only those privileges needed to carry out their task, consistent with the general principle of least privilege.

A second approach is to impose a parent-child hierarchy on VMs and to grant parents access to the contents of its children. However, this does not allow a single VM to have a cooperative arrangement with multiple management services; for this reason, a management VM and its client must be siblings or peers. For example, a client might depend on one service to keep its code base up-to-date (we call this an *environment service*), another service to authenticate users logging into the VM (we call this an *authentication* service), and yet another service to monitor its health and react to unexpected events (we call this a *monitoring service*). Each of the corresponding management VMs would be given a different set of privileges with respect to the client VM; none would have the full control over the client implied by a nested parent-child scenario as occurs in Fluke [5].

2.3 Example: PlanetLab

Much of our work is conducted in the context of PlanetLab, a global network of 400+ PCs used by researchers at over 200 institutions to develop new network services and conduct network-based experiments. PlanetLab's VMM is a modified Linux kernel, which provides both namespace and performance isolation for (typically) 150 or so users on each node, with 30–40 active on each node at any given time.

PlanetLab provides each user with one or more *slices*—a set of resources bound to a virtual Linux environment and associated with some number of PlanetLab nodes. Each PlanetLab runs a Linux kernel modified with the Vserver [13] patches to support multiple virtualised Linux environments. One difference between the Vserver approach and other virtualised environments such as Xen is that virtualisation is achieved at the system call level; consequently, each VM, called a *context* in the Vserver project, presents its users with the same kernel environment as the VMM, although with restricted privileges. The fact that each VM shares the single OS kernel turns out to have certain advantages over an approach where each VM has its own isolated OS kernel.

Each PlanetLab slice corresponds to a Vserver context, where a context represents a set of kernel objects (files, sockets, processes, etc.) that are logically isolated from all other contexts. Isolation is provided by a combination of existing kernel mechanisms e.g., the `chroot` system call that restricts a context to a particular filesystem subtree, and modifications to the Linux kernel to support tagging of objects with the context ID.

Two special contexts exist in a Vserver system: the root context, which is isolated from other contexts but typically runs in the root filesystem and has full system privileges (Linux 'capabilities'[1]), and the '*all-processes*' context, which provides full system visibility; i.e., projects the processes of all contexts into one namespace, but is typically only accessible from the root context.

A key benefit of the Vserver architecture is that each slice's filesystem subtree can be setup to closely resemble a complete Linux filesystem (modulo some differences in pseudo-filesystems such as `/proc`), which combines with the ability of a slice user to run as a restricted root user to provide a fairly thorough virtualisation of a Linux system. The restrictions placed on the slice root user are enforced by each slice being given a greatly reduced subset of the standard Linux kernel capabilities, so for example, slice root cannot change network device settings or shutdown the system. While Linux capabilities go some way towards breaking the traditional UNIX linkage between identity—what user ID a process is running as—and the operations that the process can perform, they unfortunately do not completely achieve that goal. Hence the need for a more refined solution to control access to privileged operations.

Note: to avoid confusion between the terms VM, Vserver context, domain, and slice, we use the term "slice" in subsequent sections to denote a virtual machine on PlanetLab (although strictly speaking, a slice is really a network of such VMs).

[1] We use quotes around this term since the Linux (and POSIX) notion of a capability is so different from that typically used in the OS community.

3 Architecture

Proper (from *PRivileged OPERations*) provides unprivileged VMs (slices) with access to privileged operations in a controlled manner. Although Proper was originally targeted at the PlanetLab environment it was intended from the outset to be more generally applicable, not only to other virtualised UNIX systems but ideally to systems such as *Xen* and *VMWare*.

3.1 Design Goals

The design of Proper was guided by several goals, both end-user requirements and development guidelines:

1. Proper's primary goal is to provide unprivileged applications with access to privileged operations in a controlled manner.
2. Proper must provide fine-grained access control for privileged operations, ideally down to individual object (file, port, etc.) granularity.
3. Proper must not require that the client programming model be drastically altered from the standard UNIX style.
4. Proper must be easily accessible from slice applications using a standard communications protocol.
5. Proper should not be tightly bound to a specific language.
6. Proper should be implemented as a user-space application running in the privileged (root) slice, not as a kernel extension.

The first requirement led us to consider both capability- and ACL-based designs in order to provide the necessary granularity of access control—we opted for the latter because we felt that an ACL could be more easily imposed upon the UNIX programming model of client applications (second requirement) than a capability system could. The third, fourth and fifth requirements were met by implementing Proper in C and using HTTP over loopback sockets as the communication mechanism between clients—slice applications—and the server.

3.2 Design Choices

Proper is conceptually similar to the UNIX `sudo` command, which provides unprivileged users with access to a user-specific set of privileged commands. A simple example is that of a desktop system user with no special privileges being given the ability to run the `shutdown` command in order to shutdown or reboot the system, or the ability to run `rpm` to install new system packages; note that we give no comment on the wisdom of these particular examples with respect to security.

Adapting `sudo` itself is not a practical solution in general because `sudo` is merely a binary application that runs with the `setuid` bit set so that it runs with full root privileges—in a virtualised system this becomes full root privileges within the virtual environment, so the client is still unable to interact with the host VMM. We note that extending the Linux filesystem and kernel with a `setcapabilities` bit could potentially be a solution, at least in a PlanetLab-like system, but the task of interpreting access control policy i.e., identifying who is authorised to do what, within a virtualised environment is complicated because the `sudo` command cannot trust data obtained from that environment.

Instead, we designed Proper to run as a service in the root context i.e., with a full set of privileges (capabilities), that communicates with clients in virtualised environments—slices—to perform privileged operations on their behalf.

At the highest, level Proper is simply a traditional multi-threaded RPC server that uses HTTP as a simple RPC protocol. A listener thread waits for requests to be received on a well-known port, then dispatches each request to a pool of worker threads. An important benefit of this multi-threaded approach is that requests do not have to be handled completely sequentially—if one particular requests takes a long time to complete, others are not delayed.

Once a request is received Proper checks that the client is authorised to perform the requested operation, performs the operation on behalf of that client, and returns the result as a simple text string sent as the HTTP response. In almost all cases the operation is performed after forking a process and reducing its privilege level to the minimum necessary, in case of a malicious request or implementation bug.

Much of Proper's architecture is similar to various *system-call interposition* systems [16, 7]—the use of a user-space process to interpret an access control policy, ability to provide unprivileged applications with selected elevated privileged, etc. Proper differs in two important respects: it is designed as an explicitly-invoked service rather than a transparent security layer to be imposed upon unmodified applications, and it requires no modification of the VM environment in which client applications execute. These differences mean that Proper can more readily present a uniform interface across a variety of virtualised environments, at the cost of imposing higher overhead on operation invocation.

3.2.1 Access Control

We decided early in the design process that Proper could be most readily implemented in an existing UNIX environment if an ACL-based authorisation model was adopted, since in most cases the ACL checking can just be added to the existing system implementation and fits easily into the programming model (q.v. *Linux Security Module* [24] and *SELinux* [14], or *Java*). In contrast, adopting a capability model, such as that provided by, say, *EROS* [17] or the *E* language, would have required users to embrace a radically different programming model; such an effort, while intriguing, remains a

Operation	Description
`open_file(name, options)`	Open a restricted file
`set_file_flags(name, flags)`	Change file flags
`get_file_flags(name, flags)`	Get file flags
`mount_dir(source, target, options)`	Mount one directory onto another
`unmount(source)`	Unmount a previously-mounted directory
`exec(context, cmd, args)`	Execute a command in the context (slice) given
`wait(childid)`	Wait for an executed command to terminate
`create_socket(type, address)`	Create a restricted socket
`bind_socket(socket, address)`	Bind a restricted socket

Table 1: Operations supported by Proper 0.3

project for another day.

Linux and POSIX capabilities, while poorly named, go some way towards solving the problem we are trying to address: they provide a way for the system administrator to grant only particular privileges to processes in an identity-independent manner. Unfortunately, they suffer from two significant problems: lack of operation granularity, at least in the Linux implementation; and inability to constrain the parameters used to invoke an operation. Note that the current Proper authorisation mechanism is identity-based, but relies upon a kernel feature to guarantee that the identity of a client can be determined in an unforgeable manner; getting away from this identity-based scheme is part of our future work.

Lack of operation granularity arises in the Linux capability system because only 32 bits are used to represent the set of capabilities, and the implementors, having seemingly run out of bits after allocating a fair number to appropriately fine-grained operations e.g., the `chown` operation, decided to associate whole swathes of privileged operations (even unrelated ones) with a single bit. Thus, in Linux, a system administrator wishing to give a process the ability to execute the `mount` operation also implicitly grants that process the ability to access the NVRAM device or configure the serial ports.

Parameter constraints are necessary to further enhance the granularity of access to privileged operations. For example, the `open_file` operation is one 'capability' in Proper, but we almost certainly do not want to give all users of that operation access to the same set of files. Thus, we use constraints to limit the set of files that a given client with access to `open_file` may actually open.

Proper uses a simple text file format to define its ACL: each statement in the text file associates a permitted user and a set of parameter constraints with an operation, with multiple such 'capabilities' being supported for each operation. We have found this level of detail sufficient for our current needs but plan to explore this space in the future.

3.3 Supported Operations

Our current implementation of Proper supports a small number of operations, as shown in Table 1—essentially those that service developers have asked be made available. We do not believe this list to be complete, but the fundamental nature of the operations supported suggests that it should suffice to support the majority of applications.

3.3.1 File Operations

The `open_file` operation is merely a thin wrapper around the standard `open` system call. However, since Proper executes outside of the chrooted filesystem that its clients exist in, it has direct access to all files in the system. Hence, this operation is typically used to give a slice access (usually read-only) to one or more restricted files; e.g., the root `/etc/passwd` file that maps slice names to slice IDs. This example demonstrates the need for capability constraints, since we usually grant a slice access to only a single file or directory rather than allowing it to pass arbitrary arguments to this operation.

The `get_` and `set_file_flags` operations allow clients to read and modify a small number of special-purpose flags associated with each file and directory in the filesystem. The currently supported flags are *NOCHANGE* and *NOUNLINK*. In Linux, these correspond to the *immutable* and *immutable-linkage-invert* bits supported by EXT2 and EXT3; filesystems, although other systems support similar functionality e.g., BSD has both system- and user-immutable flags.

The NOCHANGE flag was added to support efficient sharing of files between slices, as described in greater detail in Section 5.1. NOUNLINK is currently unused by any service but is used internally by Proper, as described in Section 4.3. Supporting it is essentially required due to the semantics of the corresponding Linux immutable-linkage-invert flag: because the Linux flag was added after the immutable flag was already in use it is defined to negate one particular aspect—whether the link count of a file can be changed—of that flag's behaviour, and so cannot be handled orthogonally. By not exposing these flags directly to client services, Proper can transform the orthogonal NOCHANGE and NOUNLINK flags to the underlying inter-dependent filesystem flags.

3.3.2 Mount Operations

The `mount_dir` operation supported by Proper corresponds to the Linux 'bind mount' feature, namely, the ability to attach part of the filesystem tree onto another directory; i.e., allowing the same filesystem subtree to be accessed from two directories. Figure 1 shows the motivating example for this feature in PlanetLab: a regular slice *A* wishes to have service *B* maintain its filesystem configuration (e.g., set of installed packages), so *B* uses the `mount_dir` operation to attach *A*'s root directory to its own filesystem. Section 5.1 describes how the Stork service uses Proper in this manner.

Proper also supports local—i.e., within a single slice—bind mounts, as a convenient way of providing this limited form of the `mount` system call in a safe way to clients, and a reversed form of the operation that attaches a directory from the service's slice into the client's slice (e.g., as a read-only repository of files exported by the service). The `unmount` operation exists to undo the effect of bind-mounts created by Proper.

The security implications of the `mount_dir` operation and precautions taken by Proper are discussed later in Section 4.3.

3.3.3 Exec Operations

A common requirement of unprivileged slices is that they can execute a command in another context, either another slice or the root context. Executing a command in another slice is the primitive operation that must be supported in order to enable UNIX-style remote login services, while executing a command in the root context provides a general mechanism for unprivileged slices to perform a particular privileged operation that is not directly provided by Proper (e.g., running `ps` in the 'all-processes' context).

Proper provides an asynchronous `exec` operation and a blocking `wait` operation that are typically used together in a manner equivalent to the `system()` function. The mechanics of how child processes are forked by the Proper daemon, constrained by Linux process semantics, prevent the new process from being a child of the Proper client, so the client can neither be notified by a kernel signal when the process terminates, nor use standard system calls to wait for the process to end. However, if the client forks a child process that interacts with Proper (i.e., calls `exec` and `wait`), rather than interacting directly itself, that child then reasonably approximates the behaviour of a directly forked/executed process, particularly if the process created by Proper receives appropriate file descriptors (stdin, stdout and stderr) and environment variables.

Obviously, it is important that slices are restricted in terms of the `exec` parameters they are allowed to specify: a remote login service shouldn't be able to spawn processes in arbitrary slices, while those slices authorised to execute one or more commands in the root context should only be able to execute those specific commands, usually with tightly-constrained arguments. For example, one of the PlanetLab monitoring services, SliceStat, uses the standard `top` command to acquire memory usage information for all processes in the system. Unfortunately, `top`, when run in interactive mode, allows the user to kill any visible process, so we must restrict SliceStat to only run `top` in batch mode.

3.3.4 Socket Operations

The final class of operations supported by Proper are those pertaining to sockets, specifically, creating and binding of restricted sockets. There are two classes of restricted sockets: sockets that can only be created by privileged users, such as `SOCK_RAW` and `PF_PACKET` sockets; and sockets that can be created by any user but only a privileged user can associate with certain addresses (e.g., TCP and UDP ports < 1024).

Proper supports both these classes of restricted sockets using the `create_socket` and `bind_socket` operations. Although the functionality of the latter is a subset of the former, there are practical advantages to separating the function used to bind sockets to a specific address: less application code need be modified in order to use Proper, especially if the application performs other operations, such as setting certain socket options, on a socket between creating it and binding it. For example, we created a simple wrapper library for the Apache webserver that allowed an unmodified binary application, compiled without any Proper hooks, to use Proper to bind TCP port 80 in a manner that would not have been easily possible to implement using only the `create_socket` operation.

4 Implementation

Although the architecture of Proper is relatively simple—a privileged server performing operations on behalf of unprivileged clients—there are several implementation challenges that had to be addressed to make Proper a satisfactory solution from the user perspective. This section describes our approaches to solving the most important of these challenges.

4.1 Transparent Interposition

The most obvious implementation of Proper, and one that satisfies the design goals laid out earlier, is to have the privileged server act as a proxy for client requests: when the client creates a new object, say file or socket, Proper returns a proxy object to the client that forwards all subsequent requests to Proper for application. This model works well for simple data requests (e.g., reading a file or socket), and is essentially the model adopted by systems such as CORBA [15]. However, this approach has two serious disadvantages:

1. Lack of transparency: some objects may not be amenable to being transparently 'proxied'—we note that in many ways this is analogous to the problem of virtualising system resources.

2. Performance: forwarding all object requests through the proxy to Proper imposes a significant overhead,

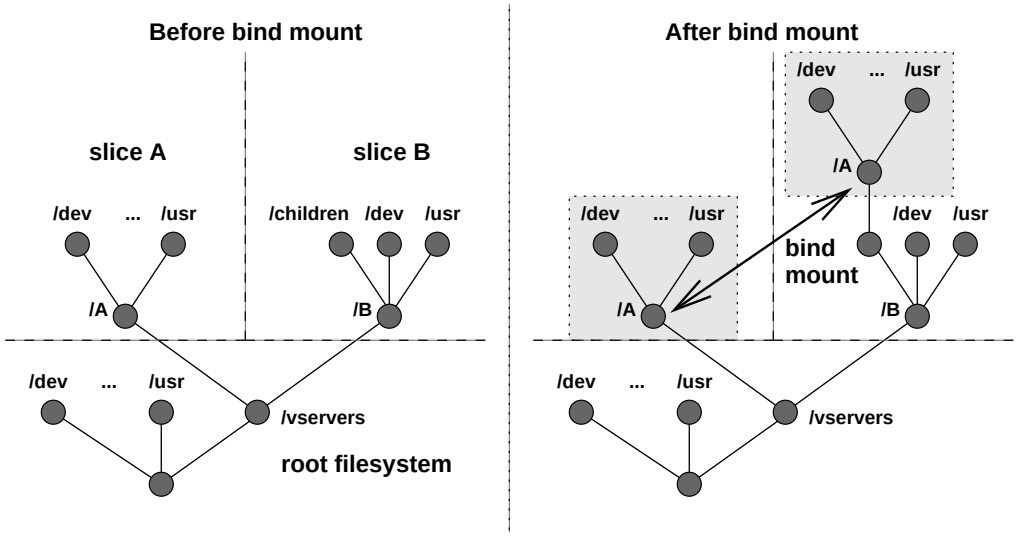

Figure 1: Bind-mounting slice A's filesystem into slice B

especially given the user-space nature of the Proper service.

Transparency is a particular problem when the objects being handled are low-level kernel objects such as files and sockets: they are not implemented in a way that makes them amenable to being replaced by proxies. A concrete example is that of opening a file: the client expects to receive a file descriptor, so it is tempting to pass back a file descriptor for a socket that is connected to Proper and which the contents of the file can be read or written over; however, if the client tries to find out the file's mode or ownership, or perform the `lseek` operation, they will be rudely awakened.

Therefore, we added two secondary design goals to the primary goals outlined earlier: *transparent interposition* and *minimal datapath interference*. Transparent interposition requires that Proper can be interposed between a client application and the underlying environment in a transparent manner; i.e., a request to open a privileged object can be forwarded to Proper without the client being aware of that forwarding taking place. Minimal datapath interference dictates that Proper should return objects to clients that connect the client directly to the desired resource (at the kernel level) rather than forcing subsequent object requests to be proxied.

Taken together, these two goals suggest that Proper should respond to client requests with exactly the same object which the client would receive if executing in a non-virtualised environment. Note that we are not so much concerned with clients determined to discover whether they are executing in a virtualised environment, but rather just presenting a sufficiently-good illusion to regular applications that they are operating in a non-virtualised world. In other words, it should be possible to insert Proper into a standard application without having to modify the application and its operation should be unchanged.

To clarify these considerations, we briefly consider each of the operation classes supported by Proper and describe how (indeed, whether) these goals are met. Note that a number of operations—`get_` and `set_file_flags`, `unmount`, `wait`, and `bind_socket`—do not return objects to the client and so are much simpler to handle.

4.1.1 File Opening and Socket Creation

The `open_file` operation was the first 'object' operation considered and thus motivated much of the transparency aspect of the Proper design. Fortunately, it is sufficiently clean and simple that the 'right' answer becomes quickly evident: just return a file descriptor opened by Proper directly to the client, which will be indistinguishable from a file descriptor that would be obtained if the client had permission to open the file directly. This is the most transparent solution possible, and also meets our goal of minimal datapath interference—subsequent read and write operations take place directly with the kernel, without either client or kernel being aware that the file was originally opened by Proper.

The key piece of mechanism that enables this solution is the ability to use UNIX-domain sockets to pass file descriptors between processes, even those executing in 'isolated' virtual machines: in this case the client creates a socket in its own filesystem subtree and passes the name of that socket as a parameter of the request to Proper, which subsequently sends the file descriptor back to that socket as the final phase of the RPC operation. This mechanism turned out to be a commonly used part of Proper; we believe that its availability in the UNIX environment makes a service like Proper straightforward, whereas other virtualised environments without a similar

```
for dir in path
    d1 = open(dir)
    set_file_flags(d1, NOUNLINK)
    d2 = lstat(dir)
    if issymlink(d2) or d1.inode != d2.inode
        return NotTrusted
return Trusted
```

Figure 2: Algorithm for verifying trusted paths

feature would not readily be able to support transparent interposition.

Creating a socket using `create_socket` is almost directly analogous to opening a file and so the same solution applies. One minor complication is the necessity to also provide a separate `bind_socket` operation, as described earlier, to support simpler interposition into client applications.

4.1.2 Inter-Slice Filesystem Access

Although the `mount_dir` operation doesn't directly return an object to the client, similar considerations can be applied since there are a number of approaches to providing the facility of one slice to read another's filesystem. For example, we could use NFS over loopback sockets to do so, but this suffers from both lack of transparency (NFS doesn't export the same filesystem semantics as a local filesystem such as EXT2) and performance overhead; we return to this point in Section 7.1.

4.1.3 Process Execution

Executing a process is the most difficult operation to transparently support, particularly given UNIX process semantics, namely the fixed child/parent process relationship. For example, Proper could more closely follow the standard `fork`/`exec` model if it was possible to 'reparent' a process i.e., attach a child of one process to another process. In Section 7.4.3 we briefly describe another possibility to increase transparency—kernel support for a `join` system call as a counterpart to `fork` that would allow one process to join itself with another.

4.2 File Descriptor Pool

As described earlier, file descriptor passing from Proper to clients is an appealing approach to transparent interposition because it gives the client exactly the same object that would have been obtained if running in a non-virtualised environment. Passing of file descriptors from clients to Proper is also a key part of the system because it allows the client to request that privileged operations be performed on objects it already holds.

While passing file descriptors from Proper to clients is relatively straightforward, the reverse direction is more complicated. The virtualised slice environment prevents the client from sending messages directly to a UNIX-domain socket created by Proper, unless that socket was created in the slice filesystem, which would in turn require that an equivalent socket be created in the filesystem of every client slice. In addition to the problems of managing a multiplicity of such sockets, Proper would also have to make sure that its usage of sockets in the client slice namespace did not conflict with the client slice's own sockets. Finally, there needs to be a way for clients to identify file descriptors that they have sent to Proper for use in subsequent operations.

We address all of these problems with a *file descriptor pool*. This is a set of file descriptors received by Proper on a single socket opened in its own namespace, with each member of the set being allocated a random 31-digit hex string as an identifier. Prior to sending file descriptors to the pool, a client application must use a special one-time method, `create_fdsock`, to open a connection to the pool—Proper allocates a new socket that is connected to the pool socket, then uses the standard Proper-to-client file descriptor passing to return that connected socket to the client.

When a client subsequently wishes to send a file descriptor to Proper (e.g., for use in `exec` or `bind_socket` operations), it performs a standard UNIX-domain `sendmsg` operation to send that descriptor. Upon receipt, Proper sends the random identifier back to the client so that the client can identify the file descriptor in later operations. In order to prevent the pool from growing too large, there is a periodic cleanup that removes file descriptors unused in a small period of time (currently 5 seconds), and file descriptors are also removed when they are actually referenced by an operation.

4.3 Filesystem Security

The PlanetLab environment places each slice and all of its associated processes in a chrooted filesystem tree, so most filesystem operations performed by Proper (e.g., `get_` and `set_file_flags`) can be safely performed just by forking a child process and using the `chroot` system call to switch into the same subtree before performing the operation; this eliminates potential problems caused by malicious symbolic links, for example. Consider a malicious client that creates a symlink of the form `../../etc` in its root directory, then requests that Proper mount the client's `/bad-etc` directory onto that directory: after following the symlinks up

through a couple of directories in the root filesystem the `/bad-etc` directory ends up mounted on top of the system's `/etc` directory. Calling `chroot` before performing the mount operation prevents this form of attack.

Unfortunately, not all privileged operations are confined to a single slice, in particular the `mount_dir` operation must often interact with two slices at once. There are general solutions to this problem, such as opening a file descriptor before using the `chroot` operation, thus allowing the opened file to be referenced in an environment where it is not directly visible, but unfortunately the syntax of the UNIX `mount` system call, specifically its use of filenames rather than file descriptors, make such solutions inapplicable.

The alternative derived for Proper is to verify that the paths used for mount do not include malicious symbolic links while simultaneously 'pinning' each path component so that a malicious client cannot replace it with a symbolic link between the verification and execution phases. This pinning is accomplished using the `NOUNLINK` flag according to the algorithm shown in Figure 2. The key insight is that setting the `NOUNLINK` flag prevents a directory being subsequently removed, opening a second time with `lstat` lets us detect a symlink that already was in place, and inode comparison[2] protects against the race condition between opening and setting the flag from being exploited.

We refer to a path where each component has been pinned in this manner as a *trusted path*. A path can only be trusted if it is rooted in a trusted path—as a base case we assume that the root directory of any slice is trusted since a slice cannot replace its own root directory.

The second security measure taken by Proper with respect to mount operations is to check against a slice-supplied ACL that the operation requester is permitted to modify the slice's filesystem (either mounting or unmounting a directory) in the requested manner. For example, when a service attempts to mount client *A*'s root directory, Proper checks that the service slice name is present in the file `.exportdir` in that directory. This gives slices control over exactly which services can access their filesystem and thus removes the need for the system default ACL to specify every permitted mount operation.

4.4 Complications in Exec

Implementing the `exec` operation presented several challenges. In addition to the difficulties in transparently emulating `fork`/`exec`, a number of other practical details must be addressed:

- Standard I/O descriptors: when running a new command it must be possible for the client to setup standard input and output correctly. For example, a remote login service that wishes to start a new shell must be able to pass a terminal file descriptor to Proper for that purpose. We use the aforementioned file descriptor pool to support this.

- Process termination: ideally the client application would be able to interact with the new process using standard operations, including using `wait()` to wait for termination. As stated earlier, this particular form of transparency is not easily supported in Proper, but a proxy process that does nothing other than call the Proper operations `exec` and `wait` can be used to achieve some degree of transparency.

- Signal handling: our goal of transparent interposition requires that the client should be able to send signals to the newly created process in exactly the same way as if it had created (and exec'd) the process itself. A proxy process that catches all signals and uses a Proper operation `kill` (currently unimplemented) to forward to the actual child suffices for many cases, but unfortunately cannot handle uncatchable signals such as SIGKILL. We continue to look for a solution to this specific problem.

The most significant practical complication arising in the implementation of `exec` was the need for `wait` operations to potentially block for an unbounded amount of time, since this means the server-side RPC implementation must be able to handle such requests. Our original choice of HTTP server library, the *Hughes Technologies* LibHTTPD library, was designed to be single-threaded with no support for deferring the result of a particular request i.e., requests must be completed in the same order they are received. Fortunately, we were able to leverage Matt Massie's *LibHTTPD++*, a multi-threaded extension of LibHTTPD, to extend Proper with the ability to handle each incoming request in a separate thread, so that `wait` requests can be completed non-sequentially. One concern is that scalability (i.e., number of outstanding request threads), may prove to be a problem, but we will address that at the appropriate time.

4.5 Evaluation of Proper Overhead

One potential concern with our current implementation of Proper is that requiring the client to invoke services using RPC over HTTP will impose significant overhead on each operation. Although we made the assumption when designing Proper that operations would not typically be used in a performance-critical manner i.e., the overhead is not important, it is of course preferable that it not be excessive.

Table 2 shows the measured performance of a pair of operations performed using Proper: `open_file` and `exec`. For each we measured the overhead for a trivial base case—opening an empty file or executing a program that does nothing—and also for a more realistic example derived from the actual operations performed by PlanetLab's users of Proper: reading a large (170MB) traffic

[2]This comparison should also take devices into account since inode numbers are not unique across block devices

Operation	Normal Latency/*ms*	Proper Latency/*ms*	Overhead/*ms*
Read empty file	1.0	13.5	12.5
Count lines in 170MB log file	270.8	282.8	12.0
Execute `/bin/true`	0.95	22.9	22.0
Execute batch-mode `top`	508	527	19.0

Table 2: Measured Latency of Proper Operations

log, and executing `top` in batch mode to get a list of all processes currently active. All results were measured using a tight loop executing only the command being tested, with a number of repetitions set to make the total execution time sufficiently large that timing inaccuracies can be ignored; the test system was an otherwise-idle 3.0GHz Pentium 4 with 1.25GB of memory, running the PlanetLab kernel derived from Linux 2.6.10.

The results show that the overhead of using Proper to invoke an operation is 12–22*ms*, depending upon the operation; a breakdown of the results for Proper show that the amount of overhead incurred by the client consists of an operation-independent value of about 7.5*ms* due to the RPC protocol, with the remainder being operation-dependent delay waiting for the Proper service to perform the operation and send back object handles (file descriptor or child process identifier). Because Proper is not involved in the application task once a file has been opened or process created, the overhead imposed is also task-independent, as shown in the table.

The measurements for the non-trivial tasks demonstrate that Proper's overhead is negligible—a few percent—in those cases; note that our chosen examples are actually less time-consuming than the operations used by our PlanetLab users: users who analyse log files are often doing something more complex than counting lines, and listing all processes takes significantly longer on real PlanetLab nodes with hundreds of processes rather than the 50 or so on our test box.

5 Client Services

Proper's current design was motivated in some part by the needs of various PlanetLab services that required access to privileged operations. This section describes a number of these services to give the reader concrete examples of how Proper is used in practice.

5.1 Stork: Service Infrastructure

Stork is a PlanetLab service that provides infrastructure to other services in the form of package management and bootstrapping functions. Stork allows users to associate software with slices, and takes care of downloading and installing the software into the slices and keeping it up-to-date. A client service specifies the set of packages it wants installed in its slice; Stork is responsible for downloading and installing those packages into the slice efficiently and securely. Stork allows package contents to be shared between slices, reducing the software footprint on a PlanetLab node. Stork also provides authentication between the Stork service and its clients, ensuring that malicious entities cannot impersonate clients or Stork. Other package management tools such as apt [1] or yum [26] exist, but they are merely tools for managing packages on independent machines, and not in a distributed slice that spans multiple PlanetLab nodes.

Ideally, the Stork service would run in an unprivileged slice. Unfortunately, this isn't possible because services must give Stork access to their file systems so that Stork can manage packages efficiently. Also, authentication between Stork and its clients is simplified if Stork is privileged. We address each of these issues in the following sections.

5.1.1 Inter-Slice File Access

Stork is responsible for downloading packages from a package repository and installing them in a client slice. Stork maintains a copy of every installed package so that each is downloaded onto a node only once, even if multiple slices have it installed. When a package is installed in a client slice the package contents must be transferred from Stork to the client. One way to do this is over a socket, but this is relatively inefficient and doesn't allow for sharing of files between slices (see the next section). Instead, Stork unpacks each package into a separate directory and uses the `set_file_flags` operation to set the NOCHANGE flag that makes the files unmodifiable. To install a package in a client, Stork mounts the appropriate package directory read-only into the client's filesystem using `mount_dir`. This gives the client access to the files in the package directory without being able to modify the directory structure.

5.1.2 File Sharing

Stork relies on Proper to share files between slices securely. Many slices may install the same packages, making it desirable to share the package contents between the slices. Of course, modifications made by one slice should not be visible to any other. The ideal solution is for each slice to have a copy-on-write version of each file; unfortunately, this functionality is not available in PlanetLab. Instead, Stork relies on sharing files read-only between slices. Files that may be modified are not shared; typically these are a few configuration files for each package. Most files are shared, dramatically reducing the amount of disk space required to install packages in slices.

Stork makes shared files read-only using the Proper `set_file_flags` operation to set the NOCHANGE flag on the files when it unpacks them. After `mount_dir`

is used to give the client access to the package directory as described in the previous section, the Stork software running in the client slice then creates hard links to the read-only files to put the files in the proper locations inside the client filesystem; writable files are copied instead of linked. The installation scripts are then run in the client slice, and the package directory is unmounted. At this point the client slice has either hard links to or copies of the files in Stork's file system. The client is prevented from modifying linked files by the NOCHANGE flag. The NOUNLINK flag is not set on the shared files; this allows the client to unlink a file and replace it with a private copy if desired.

5.1.3 Authentication

Another Stork function that is facilitated by Proper is authentication of the client to Stork and vice versa. Authentication is a common problem and there are many well-known solutions. All of them require the slices to contain secrets and prove to each other that they know the proper secret. For example, the slices could contain private keys and know each other's public keys, and the authentication protocol require that they prove to each other that each has the private key that matches their public key. This solution not only requires inserting private keys in the slices, but also a key distribution mechanism for distributing public keys.

Rather than go this route, Stork relies on Proper's operations to authenticate the client and Stork to each other. A client makes a request of Stork by sending it to a well-known port. Unfortunately, there is no guarantee that the Stork slice is listening on that port. Fortunately, the `mount_dir` operation relies on the `.exportdir` file to control who can mount a particular directory. This file contains service names, and its presence in a particular directory indicates that only the services it contains are permitted to mount the directory. PlanetLab ensures that service names are unique and cannot be forged. By putting Stork's name in the `.exportdir` file the client can ensure that only Stork has access to its file system.

The other part of the equation is for the client to authenticate itself to Stork. When Stork receives a request it has no direct way of verifying which client sent it. Without authenticating the client, it is possible for one client to convince Stork to modify another client's file system. Stork uses the following procedure to authenticate the client sending the request. Stork replies to the request telling the client to create a directory in its file system with a random name and make it accessible for Stork to mount. Stork then uses the `mount_dir` operation to mount the directory from the client's file system. If the correct client sent the request then it will have created the proper directory and the mount will succeed. Client slices do not have access to each other's file system, making it impossible for a malicious slice to create the proper directory and the mount will therefore fail.

5.2 Remote Login Service

Our second example is that of a remote login service that provides SSH access to client slices. The basic operation model of this service is unchanged from the standard SSH server: incoming TCP requests to port 22 are authenticated using a variety of methods, and a successfully authenticated client gets a connection to a new shell process running as the requested user, or slice in the PlanetLab case. One simple reason for taking this approach to remote login is that only one server can be listening on a given TCP port, so it is not possible to directly run an SSH server on the well-known port in each slice. Although it is perfectly feasible to run the SSH server in the root context, we chose to place it in a regular slice as an example of how one might create other authentication services.

The primary requirement of this service is the ability to execute a new command in a client slice. Although the Proper `exec` operation readily supports this, one outstanding problem is how clients indicate to Proper that they are prepared to allow the login service to start a new process in their slice. As discussed later (see Section 7.2), we envision a mechanism whereby a client can obtain a capability for a particular command, then give that capability to the login service for subsequent presentation to Proper. In this way the service is only able to run those commands explicitly authorised by the client slice.

5.3 Monitoring & Anomaly Detection

A key part of the PlanetLab infrastructure is auditing of outgoing network traffic in order to be able to record the source of packets that may generate adverse reactions on the destination systems: a number of PlanetLab experiments have network traffic characteristics that unfortunately trigger firewalls and IDS. Originally, this `netflow` tool was part of the root context, but a recent upgrade to the PlanetLab node infrastructure has relocated this functionality into a slice, with Proper used to provide privileged access to various network objects. Again, this enables the development of alternative auditing services.

Netflow's primary requirement is the ability to open a raw (actually `PF_PACKET` socket) that can receive all outgoing packets. Since the bandwidth of outgoing traffic on a PlanetLab node can be significant—several GB in a single day, with bursts up to 10s of Mb/s—this is one situation where having the client (netflow) be able to receive data directly from the kernel rather than having to use a proxy object is important, since PlanetLab nodes are always heavily loaded and thus we wish the monitoring service to be as lightweight as possible.

A similar service that matches Proper very nicely, but has not yet been implemented, is a intrustion detection service that sweeps through a slice's file system looking for the modification, insertion, and deletion of configuration files, binary files, and log files. Such a service, as exemplified by Seurat [25], would benefit from Proper in

much the same way as Stork.

6 Related Work

Although virtualisation has been a long-standing topic of research, and has recently seen a revitalisation in interest, there is little prior work on the subject of cooperation and communication between virtualised environments. For the most part, this is simply due to the fact that virtualised systems have not needed to address the problem—in most cases users actually desire complete isolation between VMs (slices).

A number of research projects have addressed the problem of isolation in traditional UNIX-based systems by proposing fine-grained access control at the system call level—*system call interposition*. Both *Ostia* [7] and *Systrace* [16] have a number of similarities to Proper, particularly the use of a user-space process to implement the access control policy, but require some amount of kernel code in order to interpose themselves on the system call path of arbitrary applications. This requirement reflects a difference in motivation: Proper assumes that applications are aware that they are using Proper (although shared library tricks can be used to hide this). We believe that the Proper API can thus be more readily adopted as a common interface across diverse virtualised environments.

Another direction one might look to for similar work is the development of new security architectures for existing systems, such as the *Flask Security Architecture* [18] and *Linux Security Modules (LSM)* [24]. Flask introduced the concept of a security manager as a component external to the OS kernel that implements a security policy; i.e., determines which principals can perform which operations. The LSM project enhances the Linux kernel with a large number of security hooks that can be handled by loadable modules, thus providing increased flexibility in security policy. The Flask architecture has been successfully implemented in the LSM framework by the *SELinux* [14] project.

Java's security model [8] adopts the stack inspection method [6] as a means of safely allowing unprivileged code to call into and be called from privileged code in a constrained manner, with the access policy being externally defined. This approach, conceptually similar to Proper, looks to have some interesting properties but focuses on providing a secure sandbox for subsystems within a process and is not immediately applicable to a multi-process environment.

Proper can, of course, be applied to other virtualised environments. Solaris *Zones* [19] are conceptually very similar to Linux *Vservers* in that multiple virtual environments/namespaces are created within a single OS kernel. Proper thus fits naturally into the *Zones* environment just as it does *Vservers*.

Fully virtualised systems such as *VMWare* [22], *Ensim* [4] and *Virtual PC* [21] provide users with virtualisation at the hardware level, thus allowing any unmodified operating system to be run as an application, although frequently the cost of virtualisation is sufficiently high that only a very small number of such virtual environments can be concurrently supported. In many of those systems a Proper-like mechanism is used to allow the OS running in the virtualised environment to bypass virtualisation in a controlled manner and obtain direct access to certain elements of the host system e.g., access to host system files. Unfortunately the protocol for accessing these features is typically proprietary and thus comparison with Proper is difficult.

Paravirtualised systems such as *Xen* [2], *Denali* [23] and *User-Mode Linux* [20] provide a compromise between approaches such as *Zones* and *Vservers*, running 'thin' virtualised environments on a single kernel, and the heavyweight hardware virtualisation of *VMWare* et al. Such systems require that the guest OS be modified to run within the paravirtualised environment, but the advantage over a fully virtualised system is efficiency and scalability. Of particular relevance to Proper is the support in version 2.0 of *Xen* for running unmodified Linux device drivers in a special domain and modifying each guest OS to communicate with that domain. We hope to be able to compare Proper and *Xen 2.0* in the near future.

7 Discussion

Proper was originally intended as a simple service that would satisfy the needs of a small number of PlanetLab users that wanted to implement alternative management services. However, it soon became clear that many of the design and implementation decisions raise a number of more interesting architectural questions, and open avenues for future work.

7.1 Inter-VM Filesystem Sharing

Perhaps the only form of inter-VM communication that has been studied to date is the sharing of filesystems between VMs. Kotsovinos et. al. [11] describe various filesystem mechanisms used in the *XenoServer* environment to efficiently share filesystem images between multiple Xen domains. The basic idea is that each client mounts its filesystem from a trusted domain, the *management virtual machine*, using NFS over the loopback interface. While this solution fits easily into the model of most guest OSes, a simple measurement we performed shows that loopback NFS provides significantly lower filesystem bandwidth than the bind mount facility used by Proper.

Figure 3 shows the performance measured for a number of filesystem configurations when tested with a pair of popular filesystem benchmarking tools: *dbench* and *postmark*. The four filesystem configurations compared are a stock Linux system (version 2.4.22), the PlanetLab Linux kernel (PLK, essentially also 2.4.22) using one slice bind-mounted into another using Proper, *XenoLinux* i.e., Linux (2.4.26) on Xen, using loopback NFS to share files, and PLK using loopback NFS. We added this fourth system

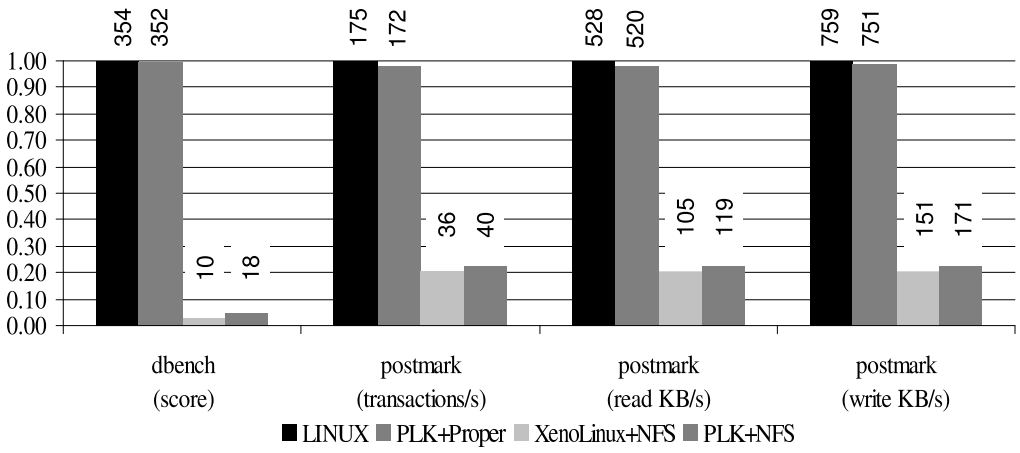

Figure 3: Relative performance of different filesystem configurations

to give us a second value for loopback NFS after private communication with the Xen team revealed the existence of a known IDE disk performance problem. All systems were tested on the same hardware platform, a *Penguin Computing Relion 1X* equipped with a 2.66GHz Pentium 4 CPU, 1GB RAM and a Seagate ST3120026A 120GB 7.2k RPM IDE disk.

The graphs show that, as expected, bind mounts incur negligible overhead over direct access to the filesystem (first two columns of each test). Loopback NFS however offers much lower performance, approximately a factor of 4 or 5 in throughput measurements and the corresponding transaction rate for the *postmark* test, and a much lower score for *dbench*. While we believe that various optimisations can be applied at the VMM and guest OS level to increase the performance of loopback NFS, the greater transparency and performance of bind mounts will be hard to match.

7.2 Capabilities and the Object Pool

During development of Proper we realised that certain subsystems in Proper have acquired a capability-like flavour; e.g., the file descriptor pool. One interesting direction for future development would be to embrace the capability model more fully, augmenting and/or replacing the ACL-based authentication and also supporting operations traditionally associated with capability systems, such as passing of capabilities from one principal to another.

One area where this might be practically beneficially is as a mechanism for a third-party slice to authorise a Proper client to perform a specific action within that slice. Examples within the currently implemented set of Proper operations include executing a process within a slice, where the third-party slice may wish to restrict the remote login slice to only being able to run a specific command, or mounting a subtree of the third-party slice's filesystem, where a capability referring to that subtree could replace the current system of client-specified ACLs.

A simple extension to Proper to support such capabilities is to generalise the file descriptor pool such it now becomes an *object pool* used to store a variety of objects. Clients and third-parties can now upload objects to the pool and receive capabilities that can be subsequently passed to other clients. One complication is that it may be desirable for certain capabilities e.g., a third-party slice's 'exec a new shell in my slice' capability, may need to persist across restarts of the Proper service; an obvious solution is to store those capabilities in a persistent database, but a more interesting, perhaps complementary, approach is to incorporate dynamic updating (see below) into Proper as a way to maintain state across upgrades of the service[3].

7.3 Port to other Virtualised Environments

A key initial goal for Proper was that it could be ported, and be useful, in virtualised environments other than PlanetLab/Linux. For example, consider how such a service might function in *Xen*: an authentication domain (domains being Xen's entity corresponding to PlanetLab slices) wishes to provide remote shell access to a number of users, each with their own Xen domain. How would Proper work in this environment?

Our initial response to this question is that it is not obvious how one would implement Xen-Proper in order to support the `exec` operation as required by this example. Possible solutions include modifying the paravirtualised kernel (i.e., the kernel running in each domain, to support creation of a process by an outside entity), or a regular process running inside the domain could be used to perform process creation on behalf of Xen-Proper.

Similarly, enabling filesystem sharing equivalent to

[3]Of course, some persistent state is always needed to handle unavoidable restarts such as node reboots and unrecoverable errors.

the 'bind mount' functionality available in the PlanetLab Linux environment is not as straightforward; one possible solution uses a low-level copy-on-write filesystem to share files between domains.

Overall, we believe that much of the ease of implementing Proper that was derived from having Proper run in exactly the same OS namespace as the clients that it interacts with would be lost in a system with a higher degree of virtualisation, such as Xen. Furthermore, certain inherent benefits of every client sharing a single OS kernel appear difficult to achieve in a one-kernel-per-domain virtualised environment. However, we believe this is potentially a very fruitful area for further investigation.

7.4 Future Directions

Although Proper has been deployed and operational on PlanetLab for some time now, there are several future directions that we plan to investigate.

7.4.1 Dynamic Updates

It is often necessary to upgrade an infrastructure service such as Proper, either to fix bugs or to add support for new operations. Although this is normally accomplished by uploading a new version of the software and restarting the server, we are looking at ways of dynamically updating the server without restarting it. Being able to do so can potentially reduce the amount of state that must be made persistent by Proper, such as the contents of the object pool, especially the mapping of capability strings to low-level objects.

7.4.2 Pool-based Memory Management

Another concern for long-running services is memory leaks, or more generally, resource leaks. The structure of Proper as an RPC server makes it amenable to a resource management strategy similar to the *region-based* memory management used in the Cyclone [9] language; a new resource pool is created at the beginning of each RPC request, and the complete pool deallocated once the request has been handled—this way, no resources allocated in the context of a particular request can leak outside that request. Of course, some mechanism must be made to allocate objects that persist beyond a particular request, but we believe this to be an approach worth investigating.

7.4.3 Kernel Integration

An explicit goal in our original design of Proper was that it be implemented as a user-space service: this eases implementation, debugging and testing but also makes it potentially easier to port to other virtualised environments. However, there are certain types of operation, such as `exec`/`wait`, that cannot easily be transparently invoked in a UNIX-like environment, for reasons described earlier. Thus it becomes necessary to contemplate the integration of Proper with the underlying kernel. Note that this doesn't necessarily require that Proper in its entirety be implemented as kernel code, although that is one, albeit unappealing, option. It just means that certain kernel modifications to aid Proper be considered.

For example, we imagine adding support for a `join` system call, that would permit a process to retain its current process ID and position in the process hierarchy, but have its system state replaced with that of another specified process (of course, the latter process must somehow indicate that it wishes to be joined to). This is essentially the same as the `exec` system call but using an existing process to create the new process state rather than a binary application. Such a facility would permit Proper to create child processes on behalf of unprivileged clients, then have the client `join` with that new process to have it attached into the appropriate place in the client's process tree, thereby allowing subsequent interactions with that child in a completely transparent manner.

7.4.4 Resource Accounting

While PlanetLab is interested in both performance and namespace isolation, Proper primarily punches holes in namespace isolation without answering the question of who gets charged for any work that is performed: the client slice or the slice providing the service. The current implementation charges the service provider, on the premise that that PlanetLab grants this slice enough resources to do its job on behalf of a multiple clients. Determining how to pass charges for work from one slice to another, or account for multiple slices that take advantage of files stored in a Stork-like slice is also a subject of future work.

8 Conclusions

As part of the PlanetLab project we found that it was necessary to give unprivileged clients running inside a virtual machine (VM) access to certain privileged operations, but in a safe and controlled manner. To meet this requirement we designed and implemented Proper, a user-level service running in the privileged root VM that performs operations on behalf of those unprivileged clients, using a simple ACL to determine which operations are permitted for each client.

We believe that a key part of this work is implementing these operations in a manner that is transparent, as far as can be reasonably achieved by a user-space application, to client programs. For example, when opening a file we return a file descriptor object to the client that is indistinguishable from that which would be given to the client by the kernel if the client had full privileges. This allows us to remove Proper from the critical path of subsequent data manipulation operations by clients.

Although brokering access to restricted objects is an important part of Proper, a more interesting challenge is how one allows one VM to directly interact with another VM, deliberately breaking the isolation between virtual machines. For example, Proper supports both execution of a process in another VM and attaching another VM's filesystem to the client VM's.

The key insight from this work is that supporting com-

munication between VMs requires a degree of support from the underlying virtual machine monitor (VMM). Systems that virtualise at the system call level, such as Linux Vservers, readily support certain forms of inter-VM communication, whereas systems like Xen that are more thoroughly virtualised are likely to require some modification to support the required forms of sharing and communication.

References

[1] Debian APT tool ported to RedHat Linux. http://www.apt-get.org/.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proc. 19th SOSP* (Lake George, NY, Oct 2003).

[3] BAVIER, A., BOWMAN, M., CULLER, D., CHUN, B., KARLIN, S., MUIR, S., , , AND AND, T. S. Operating System Support for Planetary-Scale Network Services. In *Proc. 1st NSDI* (San Francisco, CA, Mar. 2004).

[4] ENSIM CORP. Ensim Virtual Private Server. http://www.ensim.com/products/privateservers/index.html.

[5] FORD, B., HIBLER, M., LEPREAU, J., MCGRATH, R., AND TULLMANN, P. Interface and Execution Models in the Fluke Kernel. In *Proc. 3rd OSDI* (New Orleans, LA, Feb 1999).

[6] FOURNET, C., AND GORDON, A. D. Stack Inspection: Theory and Variants. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages* (Portland, OR, Jan 2002).

[7] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proc. 2004 Symposium on Network and Distributed System Security* (2004).

[8] GONG, L. *Inside Java 2 Platform Security*. Addison Wesley, 1999.

[9] JIM, T., MORRISETT, G., GROSSMAN, D., AND HICKS, M. Cyclone: A Safe Dialect of C. In *Proc. USENIX '02* (Monterey, CA, Jun 2002).

[10] KAMP, P.-H., AND WATSON, R. N. M. Jails: Confining the Omnipotent Root. In *Proc. 2nd Int. SANE Conf.* (Maastricht, The Netherlands, May 2000).

[11] KOTSOVINOS, E., MORETON, T., PRATT, I., ROSS, R., FRASER, K., HAND, S., AND HARRIS, T. Global-scale service deployment in the XenoServer platform. In *Proc. of the 1st Workshop on Real, Large Distributed Systems* (San Francisco, CA, Dec 2004).

[12] LESLIE, I. M., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P. T., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE J. Sel. Areas Comm. 14*, 7 (1996), 1280–1297.

[13] LINUX VSERVERS PROJECT. http://linux-vserver.org/.

[14] LOSCOCCO, P., AND SMALLEY, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proc. of the 2001 USENIX Annual Technical Conference (FREENIX Track)* (Boston, MA, Jun 2001).

[15] OBJECT MANAGEMENT GROUP (OMG). Common Object Request Broker Architecture (CORBA). http://www.corba.org/.

[16] PROVOS, N. Improving Host Security with System Call Policies. In *Proc. 12th USENIX Security Symposium* (Washington, DC, Aug 2003), pp. 257–272.

[17] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: A Capability System. In *Proc. 17th SOSP* (Kiawah Island Resort, SC, Dec 1999).

[18] SPENCER, R., SMALLEY, S., LOSCOCCO, P., HIBLER, M., ANDERSEN, D., AND LEPREAU, J. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. 8th USENIX Security Symposium* (WA, Aug 1999).

[19] TUCKER, A., AND COMAY, D. Solaris Zones: Operating System Support for Server Consolidation. In *3rd Virtual Machine Research and Technology Symposium Works-in-Progress* (San Jose, CA, May 2004).

[20] User-Mode Linux. http://user-mode-linux.sourceforge.net/.

[21] Virtual PC. http://www.microsoft.com/windows/virtualpc/default.mspx.

[22] VMWare. http://www.vmware.com/.

[23] WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. Scale and Performance in the Denali Isolation Kernel. In *Proc. 5th OSDI* (Boston, MA, December 2002), pp. 195–209.

[24] WRIGHT, C., COWAN, C., SMALLEY, S., MORRIS, J., AND KROAH-HARTMAN, G. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium* (San Francisco, CA, Aug 2002).

[25] XIE, Y., KIM, H., O'HALLARON, D., REITER, M., AND ZHANG, H. Seurat: A Pointillist Approach to Anomaly Detection. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004)* (Sep 2004).

[26] Yum: Yellow Dog Updater Modified. http://linux.duke.edu/projects/yum/.