



PLANETLAB

Evolving the Slice Abstraction

Larry Peterson
Princeton University

John Hartman
University of Arizona

Steve Muir
Princeton University

Timothy Roscoe
Intel Research Berkeley

Mic Bowman
Intel Corporation

PDN-04-017
January 2004

Status: Ongoing Draft.

Evolving the Slice Abstraction

Larry Peterson, John Hartman, Steve Muir
Timothy Roscoe, and Mic Bowman

January 2, 2004

1 Introduction

As the central abstraction of PlanetLab, slices initially amounted to little more than than a Unix account on set of machines. Vservers add a level of virtualization and minimal isolation, but slices remain relatively primitive. Most of the work involved in creating a complete service execution environment is currently left to the services themselves. The next step is to learn from this experience to define a richer slice abstraction. This involves defining the minimal interfaces and mechanisms needed to enable more powerful infrastructure services. These changes are in two major areas: *brokerage services* used to acquire (allocate) slice resources, and *environment services* used to create and maintain slice execution environments. For each, this document states our requirements, outlines the changes we recommend, justifies our proposal with respect to alternative designs, and illustrates how the new mechanisms might be used in one or two example scenarios.

Orthogonal to the ways in which we extend the slice abstraction is the issue of where new functionality is implemented. There are at least four possibilities: in a generic slice, in a privileged slice, in the node manager, or in the kernel.

Design Principle: *As a general rule, each function should be implemented at the “highest” level possible, that is, expecting services running in generic slices to provide a capability is preferred to requiring a privileged slice, which is preferred to the augmenting the node manager, which is preferable to a kernel-space implementation.*

Design Principle: *Privileged services should be granted the minimal privileges necessary to support the desired behavior. They should not be granted blanket superuser privileges.*

2 Slice Usage

We envision slices being used in multiple ways, possibly with different brokerage and environment services tailored to support each usage model.

Service: PlanetLab originally assumed a one-to-one relationship between slices and services, and this continues to be the dominant usage model. There may be multiple users responsible for the service, each of whom may need to log into individual VMs that belong to the slice. On the other hand, it may be the case that a slice employs an environment service to keep the service’s code up-to-date, in which case the users responsible for the service may never have need to log into the slice. Slices that support services are likely to be long-lived, and have a fairly static requirement for what nodes they want to run on; e.g., “all nodes”, “one node per site”, “all Abilene backbone nodes”, and so on. Service-oriented slices are initially associated with a site, but should the service become mature and widely-used, we expect such slices to become site-independent. This is largely for resource allocation reasons, that is, a popular service is likely to require resources beyond those allocated to an individual site.

Experiment: Experiments are much like services, except perhaps less long-lived. Experiments seem more likely than services to require that a set of users be able to log into the slice, although one could imagine users employing an “experiment management service” (i.e., a specialized environment service) to keep the experiment’s code up-to-date and control the experiment’s configuration. It also seems likely that experiments need to be able to rapidly change the set of nodes on which the slice runs, for example, to test different network topologies. Finally, experiments are more likely than services to put a burst of load on PlanetLab at predictable times, for example, just before conference deadlines. Coupled with their shorter duration, this implies that an experiment management service might batch a set of experiments, effectively timesharing PlanetLab resources among them.

Course Projects: A single slice could support all the students in a networking class, each of whom (individually or in teams) might be engaged in a different class project. Each student project could run in its own slice—just like an experiment or a service—but it often makes sense to allocate a single slice to a set of related class projects, either because there are too many students in the class to give each their own slice, or because all the projects depend on a shared software environment and managing a single copy of this environment is easier than managing multiple copies running in multiple slices.

Services and experiments are essentially the same with respect to the relationship between users and slices: all users log into a shared account on each VM using their own private/public keys. In contrast, a project-oriented use of slices requires that each user that logs into the slice be placed in their own account (i.e., be assigned a unique uid and home directory), with project teams implemented as user groups and only the course instructor given root privilege. Support for user-specific accounts could be managed by a “class project environment service” using, for example, a per-slice ssh daemon.

In all three scenarios, accountability can only be reliably done at the granularity of a slice, as opposed to individual users that have been assigned to the slice. This is primarily because PlanetLab has no control over how a slice manages users and accounts within the slice. For example, a slice can install its own ssh daemon and authorize new users without PlanetLab’s knowledge, or in the limit, it might provide an anonymous service that never authenticates anyone. In addition to the site PI—who is ultimately responsible for all the slices created for the site—we advocate identifying a single *owner* for each slice. Minimally, the owner is the first among the set of users assigned to the slice. Ideally, “slice owner” becomes a first class abstraction. Thus, PlanetLab accounts for site PIs and slice owners, even though it cannot faithfully account for individual users within a slice.

An experiment-oriented environment service that batches slice creation requests includes elements of resource allocation (i.e., is also a brokerage service), which has other implications. For example, either the centralized PlanetLab authority must allocate some fraction of PlanetLab’s global resources to such a service for re-allocation to experiments, or there must be a means by which sites can contribute some portion of their share of Planetlab to a common pool, for use by all sites that are willing to allow the service to arbitrate resource allocation (i.e., sites that are willing wait their turn in a batch queue).

3 Brokerage Services

Always a goal of the slice abstraction, we are now ready to explicitly bind resources to each virtual machine, and enforce those binding with the appropriate scheduling and allocation mechanisms. Initially, the resources bound to a given VM will be mostly controlled by a central policy, with the interface to make these bindings evolving incrementally to give PlanetLab users more and more control. We expect the first step will allow PlanetLab users to differentiate between the resources given different slices, with the ability to make hard guarantees following in later stages.

We also expect much of the work required to secure resource to fall to resource *brokerage services*, but the node manager and kernel must change to enable them.

3.1 Requirements

There are three high-level requirements:

1. It should be possible to give absolute resource guarantees to select slices. All other slices will be allocated weighted shares of the available resources.
2. It should be possible to limit slice behavior in other ways, including the TCP/UDP ports a slice can access, the number of different IP addresses a slice can send packets to, whether the slice can address machines outside PlanetLab or outside a PlanetLab site, and so on.
3. It should be possible for two different entities to control how resources are allocated: (1) the PlanetLab Consortium, which allocates PlanetLab-wide capacity to member institutions, and (2) local site administrators, which decide how to allocate capacity on the nodes they own.

With respect to the third requirement, we envision the following evolutionary path. Initially, PlanetLab runs a “bootstrap” brokerage service, called PlanetLab Central (PLC), which allocates capacity directly to member institutions (and their projects), as well as to other brokerage services for redistribution. At the same time, each node allocates local capacity to one or more brokerage services for redistribution, initially allocating 100% of its capacity to PLC for redistribution according to PlanetLab Consortium policies. Over time, as new resource brokerage services mature, it is likely that local administrators will decide to employ different, or even multiple brokerage services.

Starting with a single PLC brokerage service, there are three ways to bootstrap new services: (1) PLC gives a new service some fraction of the global capacity to redistribute; (2) PLC grants each site its share of the global capacity, and rather than assign all of its share to local projects, the site employs another brokerage service to manage (trade) its excess capacity; or (3) each node gives a new brokerage service some fraction of its capacity to redistribute. We will start with (1) and (2), which are the same from PLC’s perspective, but design for the likelihood that (3) will happen at some time in the future. As explained below, there is a well-defined point in the evolution of the technology where (3) will become feasible.

Note that there are other possibilities. For example, it is possible for the local authority to allocate capacity directly to slices, but we expect there to be an inter-

mediate service acting on the site's behalf, and so we ignore this possibility for now. It is also possible that a hierarchy of brokerage services exist, but we ignore this possibility, without loss of generality. Yet another possibility is that multiple, independent "PlanetLabs" exist, in which case the various brokerage services will need to interoperate across multiple authority domains. It seems likely that services designed to accommodate node-level clients would be better equipped to deal with this situation than centralized services like PLC, but we largely ignore this possibility for the time being.

For the sake of discussion, we assume two example brokerage services in addition to PLC: Emulab and SHARP. Emulab is a centralized service, so we expect PLC to directly grant it some global capacity, as outlined in case (1) above. It is also possible that sites will transfer some fraction of their global capacity to Emulab, to be pooled with capacity from other sites and redistributed by a batch experiment scheduler, as outlined in case (2) above. SHARP is a decentralized service, so we expect individual sites to grant it some fraction of their capacity, as outlined in cases (2) and (3) from above.

3.2 Components and Interfaces

Resource allocation involves the following four components:

- The kernel implements schedulers and allocators that enforce resource-to-slice bindings on each node. These kernel schedulers respond only to directives from the node manager. Note that the interface between the node manager and the schedulers is completely private, and not discussed in this document.
- The node manager exports an interface that allows various brokerage services to control the resources bound to each slice; we call this the *node manager interface*. Initially, the node manager adopts a trivial policy of allocating all local resources to the PLC brokerage service. Over time, the node manager will need to export a policy interface that allows the local site administrator to control what brokerage services are allowed to distribute resources on that node; this policy interface is not discussed in this document.
- PLC is a privileged service used to bootstrap other brokerage services. Initially, it is the only brokerage service that is allowed to interact with the node managers. PLC exports a GUI interface to PlanetLab users and a programmatic interface to other brokerage services; this document focuses on the programmatic variant of the PLC interface. Note that PLC also includes

a policy interface (not described) that allows the PlanetLab Consortium to control how many resources are assigned to each site and/or project.

- We expect multiple brokerage services to emerge. Emulab and SHARP are two that exist today. These services initially employ the PLC brokerage service to acquire resources that they redistribute. Eventually, they may interact directly with individual node managers.

This document primarily focuses on the PLC interface, with the node manager interface initially treated as private; that is, the node manager interface is exposed only to PLC, and hence, free to change at will. To a first approximation, the node manager interface corresponds to the `rspec/rcap` interface defined by Chun and Spalink [], that is, it supports

```
rcap ← acquire(rspec)
bind(slice_name, rcap)
```

operations, where `rspec` is a structure that contains fields corresponding to each resource. It also contains a `share` field, which is relevant to the following discussion. Note that it is the `acquire` operation, in particular, that a brokerage service must be privileged to successfully invoke.

The main reason for hiding the node manager interface is that its semantics are not well-defined at this point: how to divide resources into allocatable units is an open problem, and to compensate for this difficulty, the fields of the `rspec` are meaningful only to the individual resource schedulers running on each node. This situation is too implementation-dependent to standardize.

The evolution strategy is to transition through a sequence of PLC interfaces, $PLC_1 \dots PLC_n$, where each PLC_i grows in sophistication until it finally approaches the node manager interface, or said another way, until the node manager interface becomes well-defined. Each version of the PLC interface depends on a policy engine to “translate” the operations its supports into the corresponding node manager calls. In effect, this is an exercise of moving semantics out of the PLC policy engine and into the PLC interface until the translation from PLC_i to the node manager interface is so trivial that we can directly expose this interface to other brokerage services. At this point, the node is free to deal directly with services other than PLC since there is a well-defined way of doing so.

Since each PLC_i is a moving target, we need to accommodate brokerage services that may or may not track changes to it. For example, Emulab might be satisfied with the capability provided by PLC_1 , and hence, never upgrade to newer versions, while SHARP may track the changes at each step. We assume PLC tracks itself in

the sense that it adapts its end-user GUI to take advantage of the new capabilities provided at each stage. The key is that at each iteration, PLC_i will adopt a policy of allocating some fraction of its capacity to PLC_{i-1} , and hence, those services that still depend on earlier versions of the interface will continue to receive resources as long as they provide service to some set of users. In other words, we can think of the sequence of versions of the PLC interface as being layered on top of each other. Note that we expect there to be only a handful of PLC versions, perhaps as few as two or three.

3.3 PLC Version 1: Relative Shares

The first version of PLC will allocate relative shares, where 1 share results in a slice receiving $1/Nth$ of the available capacity and w shares results in the slice receiving w/Nth of the system's capacity. The programmatic interface for PLC_1 includes the following operations:¹

```
create_slice(slice_name, credentials)
delete_slice(slice_name, credentials)
set_owners(slice_name, owners[ ])
set_state(slice_name, boot_state, credentials)
set_resources(slice_name, resource_spec, credentials)
instantiate_slice(slice_name, nodes[ ], credentials)
```

where $boot_state = \{boot_script, es_slice\}$, and $resource_spec = \{share, duration\}$. Note that some of these operations effect only state maintained at a central PLC database, while others have the potential to effect allocations on nodes themselves, as defined below. The semantics of these operations are defined as follows:

- **create_slice**: creates a new slice with the given name. This operation effects only central PLC state; it does not cause any action to be taken on PlanetLab nodes. All other operations use this name to uniquely identify the slice. The operations can only be invoked by the site PI(s).
- **delete_slice**: removes a slice, including both central PLC state and the instantiation of the slice on PlanetLab nodes if the slice is currently running. Removing a slice with some number of shares assigned to it effects the shares available to other slices belonging to that slice. (See the discussion of shares

¹An alpha version of PLC_1 is currently defined on the PlanetLab web site. This report is a pseudo-code description of that interface. It should not be interpreted as a precise specification.

given below.) This operation can be invoked by either the site PI(s) or one of the owners assigned to the slice.

- **set_owners:** identifies a user as an owner (person that is responsible for) a slice. This operation effects only central PLC state if the slice has not yet be instantiated; it also effects per-node state if the slice already exists on some set of nodes. It can be invoked by the site PI(s) or one of the owners previously assigned to the slice. Once a slice has been instantiated, any of the owners is able to log into the slice using the PlanetLab-supported SSH daemon running at the well-known SSH port. (Note, an environment service that manages the slice is also free to run a per-slice SSH daemon and assign users to the slice in whatever way it choses.)
- **set_state:** sets the boot state for the slice; this operation is relevant to the discussion about environment services in Section 4. This operation effects only central PLC state if the slice has not yet been instantiated; it also effects per-node state if the slice already exists on some set of nodes. It can be invoked by either the site PI(s) or an owner assigned to the slice.

Unresolved Issue: *The exact form of the boot_script is still to be determined. The current working assumption is that the boot_script is a complete rc.vinit file, with PLC providing a default file if the slice owners leave this argument unspecified.*

- **set_resources:** assigns shares, for some duration of time, to a slice. If the slice has not yet been instantiated (i.e., the `instantiate_slice` operation has not yet been invoked for this slice), this operation can only be invoked by the site PI(s) and effects only central PLC state; there is no effect on per-node state. If the slice has already been instantiated, this operation effectively “renews” the slice, and can be invoked by either the site PI(s) or one of the slice owners. In the former case (invoked by a site PI) both the time and share values may be changed arbitrarily. In the latter case (invoked by a slice owner), the share can only be reduced or left unchanged, but the time can be extended up to the maximum allowed by PLC policy. In both cases, both central PLC state and per-node state is effected. (Note: changing the share assigned to one slice may effect the share assigned to other slices at the same site, as described below.)
- **instantiate_slice:** instantiates a slice, causing it to be created on the specified set of nodes. If the slice has already been instantiated, this operation has the effect of changing the set of nodes the slice runs on. To implement this operation, PLC contacts the node manager on each node included in the

slice, instructing it to create a new VM, bind it to the resources specified in an earlier call to `set_resources`, load it with the boot state specified in an earlier call to `set_state`, and then start the VM. The process of initializing a VM is discussed in more detail in Section 4.

Unresolved Issue: *Whether `instantiate_slice` is synchronous or asynchronous is yet to be determined, although it is expected to be synchronous if applied to a single node (i.e., it reports whether a verser on the specified node was successfully created.)*

The term “share” is overloaded in this discussion. First, there is the `share` argument to `set_resources`. This value is used to differentiate among multiple slices belonging to a particular site. We use the term “slice-share” to refer to this value. Second, there is a per-site share, which represents a policy decision about how much global capacity a particular site (and all of its slices) are allowed to consume. We use the term “site-share” to refer to this value. Third, there is the *site adjusted share* sent to each PlanetLab node manager. This is the value actually used to schedule resources on each node, where PLC communicates this value to each node as a field of the `rspec` structure. We refer to this as `rspec.share` in the following discussion.

The relationship between these three definitions is defined as follows. As a matter of global policy, PLC gives each site some number of site-shares (say 100) by default, and then possibly adjusts this number up or down based on how much they contribute to PlanetLab (connectivity, nodes, software, etc.). Denoting the number of site-shares allocated to a given site as S and the `share` argument given to `set_resources` for slice i as s_i , the `rspec.share` communicated to each node is given by

$$\text{rspec.share} = (s_i / \sum s_{1..N}) \times S$$

for N equal to the number of active slices belonging to the site. It is up to the site PI(s) to assign s_i (slice-shares) to reflect the relative importance of the corresponding slice. Setting $s=10$ for slice A and $s=1$ for slice B implies that A will receive $10\times$ the capacity as slice B. Note that `rspec.share` will control both cycles and link bandwidth allocated to the slice, as defined by the scheduler(s) running on each PlanetLab node. All other relevant fields in the `rspec` (e.g., assigned ports, disk quotas) will be filled in by the `PLC1` policy engine.

We assume 1 `rspec.share` results in $1/N$ th of the capacity on every node in PlanetLab. This means a slice is effectively wasting half a share if it runs on only half the nodes. We can imagine altering the policy (but keeping the same `PLC1` interface)

to let a site spend the same share on a disjoint subset of nodes. There is a lot of room for tweaking the policy without changing the version 1 interface.

Evolutionary: *It will be important to provide facilities by which users can determine how much actual capacity a slice typically receives for each share assigned to it. We hope this allows us to identify one or more “canonical shares”, with subsequent versions of PLC allowing users to reserve a canonical share for their service. Later versions might then allow some services to request hard guarantees. We expect the interface to finally evolve to the point that it accepts `rspecs` and returns `rcaps`. Once in place, individual nodes are free to expose this interface to allocators other than PLC.*

4 Environment Services

Today, slices boot with a set of SSH keys and virtually no other packages installed. A non-vserver SSH daemon allows remote access to the slice. All responsibility for setting up the programming environment in which a service runs, and keeping that environment up to date over time, is left to the service provider. We expect a set of environment management tools—so called *environment services*—to emerge. To enable them, we need to extend the slice abstraction to provide the necessary hooks by which these services are employed.

There are several requirements:

1. The infrastructure must provide the minimal functionality to allow a newly created slice to be provisioned and controlled by a service built for this purpose.
2. Multiple environment services may exist simultaneously. The infrastructure must provide these services with the hooks they need to access the trusted functions required for ownership and access control management.
3. An environment service should be able to share files between it and its clients. This implies being able to combine duplicate copies of a file used by multiple clients.

For illustrative purposes, we assume Stork is an example environment service throughout the following discussion,

4.1 Components and Interfaces

Environment services are privileged, and must be flagged as such when created. In the first version of PLC, a slice that is to run an environment service informs PlanetLab of this fact through an out-of-band channel, and the privilege, denoted `SLICE_ES` is enabled on the slice using the administrative interface to PLC. This designation gives the environment service two abilities: to specialize the boot sequence of its clients, and to share files between itself and its clients.

Specializing a client's boot sequence is handled by allowing a boot script to be specified as part of the client slice's boot state. The boot script is stored in the slice database at PLC, and distributed to the node manager when the slice is instantiated. The node manager then runs this script when the vserver boots, after running any node manager-specific actions.

Evolutionary: *The boot script should be signed to ensure its integrity between when it is provided to PLC and when it is run. Nothing about the current slice infrastructure is currently secure enough to warrant doing it immediately.*

An environment service must also be able to share files with its client slices in a protected fashion. The environment service should be able to create and modify files, and the client should be able to delete, but not modify, files. This allows the environment service to share files with its clients without worrying that the clients will modify them. A clients should be able to delete its link to a file so as to replace it with its own version of the file. Two things must happen for this sharing to work. First, sharing a file between vservers requires access to both vservers' namespaces. Second, the environment must be able to write-protect the files that it shares with the clients. Solutions to both of these issues must exist within the existing Linux infrastructure. There are several options for sharing a file between vservers:

Slice Adoption. The node manager supports a function of the form

```
adopt(slice_name, rootdir)
```

that causes the root of the named slice to be moved into the specified rootdir in the invoking slice, typically that of the environment service. This gives the environment service access to the client's entire namespace. Adoption is allowed only if the caller successfully set the `SLICE_ES` attribute when it was created,² and the client service specified the invoking slice as its envi-

²For now, we assume this attribute is set by the PLC administrator using an administrative interface.

ronment service when it was created. The downside of this approach is that the environment service has access to the entire client namespace.

Export/Import. The node manager provides functions that allow one vserver to export a file or directory, and another vserver to import it. This option allows for finer grain sharing than slice adoption. Some form of access control is used to limit access to exported files or directories.

Absolute Pathnames. Pathnames within a vserver are relative to that vserver's root, making it impossible for a vserver to name a file or directory in another vserver. In this solution the node manager provides namespace functions that use absolute pathnames as parameters; e.g. "link stork:/packages client:/packages". The node manager has access to all vservers' namespaces so it is able to perform these operations. An authorization mechanism similar to slice adoption is needed to prevent unprivileged vservers from invoking these operations. This mechanism is likely to be complicated, and the overhead of invoking individual directory operations in this manner is unknown.

The current choice is to provide an export/import mechanism. A slice makes a directory hierarchy available for export by creating a file called `.export` in the directory. The file contains the names of the slices that are allowed to import this directory. The syntax and semantics of this file will likely evolve over time. The node manager provides a function

```
plnm_mount_dir(char *client, char *dir, char *mntpoint)
```

that causes the directory `dir` in the client vserver `client` to be mounted in the calling vserver at location `mntpoint`. This gives the caller access to the client's exported directory via `mntpoint`.

Files shared between clients and environment services must be protected from modification by the clients, either accidental or otherwise. There are several possibilities:

Read-Only Mount. The environmental service exports a directory that the client can only import read-only. There are two problems with this approach. First, the entire directory hierarchy is imported read-only, which means that clients cannot delete files and replace them with their own version. Second, it currently is not supported by Linux (`mount -bind -r` does not work). It is possible for the environment service to export the directory using NFS and for the client to then import it read-only, but this will have significant performance implications.

Copy-on-Write. A file is shared until written, at which time the writer gets its

own copy. Clients share files copy-on-write, the environmental service does not (its modifications are visible to the clients). Linux does not currently support copy-on-write. Also, it isn't clear what to do about directories—if a client adds a file to a directory (thereby writing it) does it get its own copy, and if so, are files subsequently added to the directory by the environment service invisible?

Immutable bits. Linux provides immutable bits for files that can only be manipulated by root. Setting both of these bits on a file makes the file's contents immutable, but not the name. This allows a client to make a link to a file without being able to modify the file. The immutable bits are preferred way of sharing files between vservers. Manipulating the bits requires the Linux `CAP_LINUX_IMMUTABLE` capability, which vservers do not currently have. The node manager provides the following functions for manipulating the immutable bits:

```
#define PLNM_FILE_DATA_IMMUTABLE 0x0001
#define PLNM_FILE_LINK_IMMUTABLE 0x0002
plnm_set_file_flags(char *file, unsigned flags)
plnm_get_file_flags(char *file, unsigned *flags)
```

These routines can only be invoked if the slice was designated `SLICE_ES` using the administrative interface to PLC.

Reserved Users. The UNIX file system provides a well-known mechanism for protecting files. Files belong to a user and a group, and different permissions can be specified for the owner, group, and everyone on each file. Suppose a shared file is owned by the Stork user, and is only writeable by the owner. Ideally, this would prevent other vservers from writing the file. Unfortunately, file system permissions do not mesh well with the vserver security context so this doesn't work. The root in the client vserver can simply `setuid` to Stork and modify the file. The solution is to incorporate the security context into the file permission checks, preventing the clients from writing the file either as root or as Stork. This is likely to be a substantial implementation effort.

Evolutionary: *Eventually, a slice will be allowed to specify the `SLICE_ES` privilege as a parameter to the `set_resources` operation. If `SLICE_ES` is set and the credentials permit it, the slice is registered as an environment service, and granted additional privileges as appropriate. It may make sense to generalize privileges to capabilities and grant capabilities based on credentials. In this scenario, slices would specify*

a set of desired capabilities (both Linux and PlanetLab-specific) as a parameter to the `set_resources`. The capabilities would be granted if the credentials allow.

Unresolved Issue: *The exact form of inter-VM communication on the same node is not currently specified. Clients need to communicate with the environment service, and the environment service needs to communicate with the node manager, and do so efficiently and securely.*

4.2 Example Environment Services

This section describes two example environment services. One is modeled after Emulab, which allows experiments to be set up across a set of PlanetLab nodes. Emulab specifies a simple boot script with each slice it creates. This script does a `wget` to pull down a tarball from an Emulab site, unpack it, and initialize the experiment. The Emulab service also installs a per-slice SSH daemon, and sets up distinct user accounts within the client slice. In this example, the environment service does not need to set the `SLICE_ES` attribute and it does not want to share files between slices.

Note that one potential problem with this approach is that it assumes the `wget` is successful, which may not be the case for an Internet-scale experiment. One fix is for the boot script to schedule a cron job that repeats until the tarball is successfully downloaded. Alternatively, the Emulab service could run in its own slice, allowing the client slice to retrieve the tarball locally.

A second example environment service, called Stork, illustrates the value of the interfaces just described. Stork assumes all software it manages is packaged in rpms, it runs as a service in its own VM on every node, users (client services) put the rpms they need on a central Stork repository, and the user creates a “package set” containing a list of these rpms.

When a user creates a slice, he or she specifies a Stork-provided script (`vc.init` file) as part of its boot state. When each VM is initialized, the script exports a package directory, then contacts the Stork service running on that node using a local, inter-VM communication mechanism. Stork imports the package directory and populates it with the packages needed by the client. These packages include package manipulation commands, a Stork daemon, a Stork initialization script. Control is then passed back to the new slice, which executes the full Stork boot script.

The Stork service is responsible for downloading and unpacking rpms as necessary, and deleting them when they are no longer needed. Stork likely uses a modified version of apt to do this, and works as follows. First, Stork unpacks a package with `/packages/package_name_version` as the root (relative to Stork's root). The immutable bits are used to protect the contents of this directory from the clients. Stork used the `plnm_set_file_flags` function to set these bits on the files it unpacks. Note that Stork must allow conflicting packages to coexist. The root change above will avoid file conflicts, but changes may be necessary to the package database.

Second, Stork unpacks the rpms into this tree, but does not run any of the installation scripts. Instead, each client has a set of Stork tools for manipulating packages, and its own package database. These tools interact with the Stork service to create links the files in Stork's package directory to the directory exported by the client (or make copies as appropriate). The client then runs the installation scripts and updates its local package database. Package signatures are also checked if necessary. Package removal consists of deleting files and links as appropriate.

Unresolved Issue: *How to deal with identical packages but signed by different parties.*

Third, the Stork service provides an interface that allows it to remotely administer packages. It does this by installing a Stork daemon into each client slice. The daemon can be configured by the service to update the package set periodically. It also accepts commands from Stork to install, remove, and update packages. This allows Stork to force services to use a particular package, or more important, to stop using a particular package. If the daemon dies, Stork can still prevent a package from being used by simply deleting it from the shared package area.

Evolutionary: *For simplicity, the initial Stork release will only share packages between clients and Stork. Clients will use rpm to unpack their own copies of the package contents, and vunify will be relied upon*