Slice Creation and Management

Brent Chun and Tammo Spalink

August 20, 2003

1 Introduction

Slices are the central resource allocation abstraction in PlanetLab. This document establishes the vocabulary needed to talk about slices, and defines an architecture for dynamic slice creation on behalf of principals. This is a working document which contains a proposed interface specification. This document is meant to provide additional detail for the architecture described in PDN-02-005.

This document describes *mechanisms* for resource allocation and policy implementation, but not *policy* itself. Policy is the decision of which slices should be granted access to which resources. The goal of this document is to describe a set of general mechanisms and tools which can be used to efficiently facilitate arbitrary policies imposed by higher levels. Different nodes, and even different sets of resources within a node, may be governed by different policies.

The lowest level in the PlanetLab resource allocation infrastructure is the *node manager*, which presents an uniform interface at all nodes. This node manager interface is designed to be very simple and thus should only change infrequently. More complex aspects of resource allocation are implemented at higher levels as special trusted *infrastructure services*. This document completely specifies the node manager interface, and provides two example infrastructure services that use this interface. The first of these, the PlanetLab Central service, is the first service to run on any node of the system and is used to bootstrap other services.

2 Node Architecture

The foundation of the PlanetLab resource allocation infrastructure is the node manager. The node manager interface is a uniform interface for communicating re-

source usage policy to a node, available at every node. The initial implementation of the node manager is a thin service layer above an existing operating system, but in principle should be thought of as presenting a system call interface.

Node managers do not make any policy decisions, they simply provide a resource access and policy enforcement interface. Policy is imposed by infrastructure services, and a set of privileged node managers operations are available only to these trusted services.

The node manager interface is not remotely network accessible, a specific node manager's operations are only available to services executing on that node. Communication authentication and encryption operations (SSL, SSH, etc) are too complex to support in the node manager. It is also likely that these mechanisms should be optimized to match the usage profile of the system. Infrastructure services provide implement these mechanisms, allowing for competing implementations and evolution over time. All nodes are bootstrapped with at least one infrastructure service which assists other services in contacting remote node managers by being widely deployed and acting as a proxy. In further defense of this design decision, we assert that functionality which can be safely and efficiently left out of a low level, and moved upward, should be (reminiscent of the end-to-end argument [5]).

2.1 Resource Distribution

Initial distribution of resources to services involves policy decisions and remote communication, and is thus implemented by infrastructure services. Once non-infrastructure services acquire resources from infrastru cture services, they are free to trade these with one another. Rights to use specific resources are represented by resource capabilities (rcaps), as described in Section 2.1.2.

Each node manager tracks both the set of resources available, and a mapping between dedicated resources and capability *rcap* datastructures that act as access promises. Infrastructure services can acquire these capabilities by presenting and node manager with a specification of a subset of the remaining available resources using *rspec* datastructures. In return the node manager will create a new rcap and remember the mapping between the two.

The node manager makes the following operation available to infrastructure services:

```
acquire(rspec) -> rcap
```

A summary of this and other node manager operations, including datastructure details is available in Section 2.7.

2.1.1 Resource Specifications

The rspec is used to describe access privileges (privs) to resources over time. Each rspec can contain a list of reservations, proportional share values, or upper bound limits for any number of different resources available on a specific node. An rspec does not include location information; thus a more sophisticated datastructure or multiple rspecs are needed to describe the resource needs for services that span multiple nodes. Each rspec also has interval information to describe the time window over which all of the contained access privileges apply.

When used for slice resource provisioning, rspecs generally include only reservation and proportional share privs. Upper bound privs are used to leverage resource schedulers to control slice behavior and are explained in more detail in Section 2.4.

Reservation privs provide guaranteed resource bounds, meaning that exactly the specified share of resources will be available during the time window. For proportional share best effort tickets, the resources available to the virtual machine during the time interval depends on the share value relative to other active privs.

Details on the rspec datastructure are presented in Section 2.7. It is expected that as the underlying resource scheduler evolves, additional types of privs will be added and existing ones refined to improve expressiveness and granularity.

2.1.2 Resource Capabilities

Node managers provide access to allocated resources in the form of rcap datastructures. The rcaps are simple 128 bit opaque values, and knowledge of an rcap value implies access to the associated resources. Because the space of possible rcap values is very large, it is improbable that any adversary is able to guess a valid rcap value and thereby hijack resources allocated to other services.

At the node manager, the rcap serves as an index into a table of rspecs. The content of the rspec, and thus the form factor of the associated resources, is a policy matter and thus was provided by an infrastructure service during an acquire operation. The node manager will create rcaps for any rspec that matches available resources, but it practice it may be desirable for these rspecs to be standardized. For example, to simplify support for resource markets it may be important that rcaps resemble

standardised commodities spanning fixed length short time intervals. This would allow for a liquid market where prices can track demand fluctuations at hourly or daily granularities. It may also be important for bounding node manager state, that infrastructure services acquire resources only within a bounded time horizon.

2.2 Resource Binding

Slice creation is really the process of binding the resources associated with an rcap to a virtual machine. To incorporate node resources, a slice needs a virtual machine. Without an rcap that includes a virtual machine resource, other rcaps are useless. Similarly, having more than one virtual machine resource per node at one time is also unnecessary.

Resources may be bound to a slice using the following node manager operation, which is available to all local services as well as infrastructure services:

```
bind(slice_id, rcap)
```

The process of determining which resources should be part of a slice may be an ongoing one, especially for persistent slices. Both the needs of the service and the state of PlanetLab may change over time, and the slice management mechanism must allow adjustment accordingly. For example, network links may become too congested to allow effective use of certain nodes, nodes may simply fail, or the demand for the service may greatly exceed the initial expectation of the principal. In all of these cases adjusting the resources that make up the slice becomes important. Slice may need to be very dynamic and creation may often be an iterative or even ongoing process.

2.2.1 Slice Namespaces

All node manager operations are performed on behalf of users called *principals*, which have entered into a legally binding usage agreement with the PlanetLab administration. During this process they are authenticated and assigned a globally unique 32 bit principal_id value.

Each principal may create and be responsible for many slices. The slice_id datastructure is a tuple of principal_id and slice_tag, where the tag is a XXX bit opaque value that can be chosen by the creating principal. This divides the slice_id namespace evenly across principals.

This document discusses actions as being performed by services. This indicates that operations may be triggered directly by software running in a slice belonging to the service, or by users responsible for the service who have authenticated themselves through an outside interface. Outside interfaces can include custom interfaces provided by the service, or those provided by infrastructure services.

2.2.2 Virtual Machine Creation and Bootstrap

A virtual machine is created for a slice at the start time of the first bound rspec which contains a virtual machine resource. At least one virtual machine resource must be bound to a slice at all times during the lifetime of the slice. Once no virtual machine is bound to a slice, no other resource bindings have effect. This means that the resources and state associated with the slice may be reclaimed and reused. Node managers handle this charitably in a best-effort manner. Unused virtual machine state is retained as long as the resource requirements of doing so do not conflict with the needs of active services, at which point the resources are reclaimed.

Once a virtual machine is created, the service must have some way of contacting it to upload software state and begin execution. A freshly created virtual machine contains a minimal file system image. After creation, a standard bootstrap script provided by PlanetLab central is run in the virtual machine which prepares it for access by its users. In the current implementation, this script contacts the local PlanetLab central infrastructure service to acquire a port number and SSH public keys for the slice's users. It then starts an `sshd` configured to allow access for those keys. This `sshd` allows the service to bootstrap the slice with its own data, and is free to terminate and remove the server and the script once it has gained access.

2.2.3 Direct Node Manager Access

Although a number of node manager operations are available only to infrastructure services, the bind operation and a few others are available to any local slice.

The node manager interface is quite primitive and is tailored to support development of efficient infrastructure services. However, operations are made available to other slices where possible to allow sophisticated slices to optimize by communicating with local node managers directly.

The bind call is treated specially by the node manager. It is available only to infrastructure services and to slices which have the same creating principal as the target slice.

2.3 Manipulating Capabilities

Once a service has acquired one or more rcaps from node managers and before these rcaps are bound, there are no architectural restrictions on their exchange between services. To facilitate transactions involving rcaps, the node manager provides the following manipulation operations which are available to all local slices:

```
reissue(rcap) -> rcap
split(rcap, rspec list) -> rcap list
merge(rcap list) -> rcap
```

The reissue operation is useful to re-establish the rcap as a secret. Any possessor of an rcap may use the reissue operation to exchange it for a new rcap with the same associated resources. The old rcap is voided, ensuring that the holder of the new rcap is the only user with access to the resources.

The split operation uses the rspec list argument as the basis for how to subdivide the rcap argument. If the provided rspecs do not sum to match the original rcap's rspec, the operation will fail.

The merge operation takes a set of rcaps and merges them into a single rcap which represents the aggregate resources of the individual rcaps. Each of the aggregated rcaps must have the same time window. This argues for choosing standard time intervals.

2.4 Imposing Policy

The node manager is the ultimate regulator of how many resources are available to specific slices, regardless of which resources are actually bound to the slice. There are a number of possible reasons why a slice may receive less than the expected quantity of resources.

Many nodes are donated by parties that have specialized usage restrictions on their equipment beyond those imposed by the general PlanetLab usage agreement. These nodes may impose limits on slice behavior and even blacklist slices or even principals. For example, a service used as a pornography distribution mechanism may be legal and accepted at some sites but not at others. In such cases nodes have

the right to isolate their resources from certain services. It is in the best interest of restrictive sites to widely advertise their limits, because it avoids the overhead of services acquiring resources for nodes they would be unable to use and the possibility that those resource may fall idle. Policy may be enforced at either the granularity of principals or slices.

In addition to persistent node usage policies, the node manager supports temporary disciplinary action. When software in any slice belonging to some principal violates resource usage agreements, either that specific slice or all slices of that principal can be potentially targeted for temporary restrictions. These restrictions may be tailored to meet the specific problem using the flexibility of the rspec datastructure.

The node manager makes the following operations for slice resource usage control available through infrastructure services:

```
limit(principal_id set, slice_id set, rspec)
unlimit(principal_id set, slice_id set)
```

The set arguments in the limit operation are used to indicate which slices should be affected by upper usage limits described in the rspec. The sets may be empty, a list, or a wildcard to indicate the universal set. The operation will affect the maximal set of slices described by the arguments. The unlimit operation removes any previous limits on a specific set of principals and slices.

Both limit and unlimit are infrastructure service only operations because deciding which principals may affect which slices is a policy decision.

2.5 Inspection and Logging

All invocations of node manager operations are logged to a persistent storage location visible to all slices along with a timestamp and the slice_id of the caller. If storage resources for logging are exhausted, the node manager will not accept additional transactions.

In addition to the log, the node manager makes internal state about slices available to all local slices with the following operation:

```
inspect(principal_id set, slice_id set) ->
  (principal_id, slice_id, rspec) list
```

The set arguments are treated similarly to those of the limit operation and state is returned for all matching slices. The respchain

2.6 Infrastructure Service Optimizations

Certain scenarios may arise where the reissue node manager operations may be invoked so often as to present an overhead burden. One example is in a very liquid resource market where rcaps change hands many times due to speculation before they are finally bound to a slice. Assuming that the node manager is contacted for each transaction to reissue, and possibly even more often to merge and split rcaps to create new commodities, the resulting message traffic volume may be undesirable.

There are a number of solutions to optimize this communication. One alternative is to have a trusted party acquire and hold rcaps, this allows the items traded be rcap references. This strategy adds a layer of indirection, a common systems approach. Another approach is to build a cryptographic security and trust infrastructure for rcap transfer.

2.7 Interface Summary

The initial node manager implementation is visible as an XML-RPC server listening on a well known TCP port. The operations it supports and and explanation of argument structure follows.

Node manager operations:

```
bind(slice_id, rcap)
reissue(rcap) -> rcap
split(rcap, rspec list) -> rcap list
merge(rcap list) -> rcap
inspect(principal_id set, slice_id set) ->
  (principal_id, slice_id, rspec) list
```

Available only to infrastructure services:

```
acquire(rspec) -> rcap
limit(principal_id set, slice_id set, rspec)
unlimit(principal_id set, slice_id set)
```

Datatypes:

```
principal_id : 32 bit opaque value
slice_tag : XXX bit opaque value
slice_id : (principal_id, slice_tag)
rcap : 128 bit opaque value
```

```
time : 64 bit natural number // seconds since the epoch
rspec :
  (time_start : time, time_end : time, priv list)
priv :
  vm_upper_bound(count) |
  vm_resvn() |
  outbound_ip_bandwidth_upper_bound(ip_addr_set, bps) |
  outbound_ip_bandwidth_resvn(ip_addr_set, bps) |
  outbound_ip_bandwidth_prop_share(ip_addr_set, share) |
  inbound_ip_bandwidth_upper_bound(ip_addr_set, bps) |
  inbound_ip_bandwidth_resvn(ip_addr_set, bps) |
  inbound_ip_bandwidth_prop_share(ip_addr_set, share) |
  ip_flows_upper_bound(ip_addr_set, count) |
  ip_flows_resvn(ip_addr_set, count) |
  ip_flows_prop_share(ip_addr_set, share) |
  outbound_ip_flows_upper_bound(ip_addr_set, count) |
  outbound_ip_flows_resvn(ip_addr_set, count) |
  outbound_ip_flows_prop_share(ip_addr_set, share) |
  inbound_ip_flows_upper_bound(ip_addr_set, count) |
  inbound_ip_flows_resvn(ip_addr_set, count) |
  inbound_ip_flows_prop_share(ip_addr_set, share) |
  udp_ports_disallow(int_set) |
  udp_ports_prop_share(share) |
  udp_ports_resvn(int_set) |
  tcp_ports_disallow(int_set) |
  tcp_ports_prop_share(share) |
  tcp_ports_resvn(int_set) |
  fd_upper_bound(count) |
  fd_prop_share(share) |
  fd_resvn(count) |
  disk_space_upper_bound(kb) |
  disk_space_prop_share(share) |
  disk_space_resvn(kb) |
  memory_pageable_upper_bound(kb) |
  memory_pageable_prop_share(share) |
  memory_pageable_resvn(kb) |
  memory_pinned_upper_bound(kb) |
  memory_pinned_resvn(kb) |
  cpu_upper_bound(percentage) |
  cpu_prop_share(share) |
```

```
  cpu_resvn(percentage)
ip_addr_set : [CIDR-prefix or ip_addr] list
count : integer
bps : integer  // bits per second
share : integer  // units of virtual currency
int_set : [integer or integer range] list
          // eg (1, 3-5, 7-9, 88, ...)
kb : integer  // kilobytes
percentage : integer  // 0-100
```

3 PlanetLab Central Infrastructure Service

PlanetLab Central (PLC) is the first service to run on a node after bootstrap, and it maintains an active slice on all nodes of the system. This section serves both as a specification of the PlanetLab Central service, and as an example use of the node manager interface.

The PLC service will leverage existing infrastructure and maintain a central database of principles, slices, resource allocations, and policies. Each node will host a slice, specific to the PLC service, which facilitates VM management and resource allocation by issuing requests to the underlying node manager. Analogous to software updates, each PLC resource manager will periodically poll a central database, downloading a file with the latest node state update information, and perform appropriate actions (e.g., create/delete slice, bind resources to slice, etc.). This approach is simple, provides clear benefits, is based on a well-known solution, and will serve as a concrete point of reference for competing systems in the future both in terms of robustness and performance.

3.1 Principals and Accountability

The PLC service will leverage the existing infrastructure for managing principals and providing accountability. PLC principals are users who have been authenticated out of band by PlanetLab central and have signed the PlanetLab Acceptable Use Policy agreement, corresponding to existing PIs.

To review existing mechanisms, all users must register their identity and upload an SSH public key through the PlanetLab web site. Users are either principle investigators (PIs) associated with contributing PlanetLab sites or users who work

under a principle investigator. Users authenticate themselves using a login and a password. PIs may authorize users to access a slice by associating the user's SSH public key with the desired slice, thereby enabling access to all of the slice's nodes. All authorization decisions are made through the PlanetLab web site and delegated based on a hierarchy of responsibility. PlanetLab central directly authorizes principle investigators. Each principle investigator, in turn, authorizes local users (e.g., graduate students) and assumes responsibility for them.

Leveraging PlanetLab central provides two key benefits. First, it allows us to build an initial resource management system that provides useful functionality and is as simple as possible. Second, it allows us to leverage an existing infrastructure for managing principals and delegating authorization that has worked quite well to date. We will use the existing authorization hierarchy to authorize all privileged actions in the resource management system. These actions include creating slices, deleting slices, assigning principals to slices, and partitioning global PlanetLab resource allocations, of which each site gets a fraction of the total pool.

3.2 Dynamic Slice Creation and Maintenance

Dynamic slice creation will be controlled centrally through the PlanetLab web site. Users will authenticate themselves using their login and password. Once authenticated, principals are able to create slices, or authorize other users to do so on their behalf.

The web interface will provide each user with accounts and currency they can use to acquire resources. Initially, each user will have two accounts, one which tacks virtual machine rights tokens and another for proportional share credits. Credits and tokens will be added to the accounts of principals at regular time intervals, based on an undisclosed algorithm which tries to provide each principal with resources commensurate to their contributions to PlanetLab. These contributions might be based on factors such as providing valuable shared services, hardware or software contributions, and adherence to the PlanetLab acceptable usage policy. Credits and tokens expire if they are not used.

To create a slice using the web interface, a user must allocate one virtual machine token for each node to be included in the slice. The user must then allocate proportional share credits to each resource category to indicate what share of the user's resource rights are to be available to that slice. Once such a token and credit allocation has occurred, it will persist and new tokens and credits will be used to sustain the slice as they become available.

3.2.1 Hierarchical Proportional-Share Resource Allocation

In addition to using tokens and credits to create slices, users may transfer them to be used by others. Principles at each site can divide their initial allocations in a hierarchical proportional-share fashion. The proportional-share hierarchy will mirror the authorization hierarchy already stored in PlanetLab central's user database. In addition, at each node of the hierarchy, a user can assign some fraction of its resource assignment to its slices. For example, a given principal might allocate 30% of its allocation to its slices and allocate a 70% proportional-share allocation to users authorized by that principal. Each node in the hierarchy essentially creates a new currency, similar to those used in lottery scheduling [7].

Hierarchical, proportional-share scheduling will initially be applied to both CPU and network bandwidth. Longer term, we could extend this to other scarce resources such as physical memory (e.g., using Waldspurger's min-funding revocation algorithm [8]) if need be. During contention, proportional-share scheduling allows sites to obtain resource allocations commensurate to their contributions to PlanetLab. Of course, we do not expect all principals to always be using their entire resource allocations all the time. Proportional-share scheduling allows these unused allocations to be used by active services, thereby improving utilization. Adding hierarchical scheduling provides additional control which preserves the compensation for valuable contributions. For example, a user at a site that makes valuable contributions to PlanetLab can obtain that entire site's allocation if all other users from that site are idle.

3.3 Administrative Policies

The PLC web interface will provide administrators with access to the policy enforcement mechanisms at each node. PlanetLab administrators will be able to set policies for nodes, slices, and principals that apply to resources allocated by all resource management systems. This interface will be similar to that provided by the node manager limit operation, extended to allow specification of which nodes to target.

This policy interface will be available to all users, providing them with disciplinary control over their own slices. Administrators are granted additional scope over all slices at specific sites or the entire network to match their domain of responsibility.

Example policies on nodes might include upper bounds on the amount and type of network traffic a node can generate. Policies on slices might include limits on what

slices are capable of doing, such as whether they can send ICMP ping packets. Finally, policies on principals might include limits on what a particular principal can do in general. For example, there might be a policy that states that a particular principal (and all users below that principal in the associated user hierarchy) can only create slices that send traffic to nodes within PlanetLab.

3.4 Periodic Node Synchronization

PLC resource managers will periodically poll PlanetLab central for the latest slice, resource allocation, and policy information. This information will be generated on a per-node basis, either dynamically (e.g., using PHP) or by distributing a set of files (e.g., named by node IP address). Each PLC resource manager will obtain this information over HTTPS by downloading a signed, timestamped file that contains all the necessary information for all slivers on that particular node. This information includes a list of slivers that should be present on the node, hierarchical resource allocation information for each sliver, and policy information at the node, slice, and principal level. For each sliver, the resource manager will then take the sliver's resource allocation, apply relevant policies, and obtain an appropriate capability which is then bound to a sliver.

In the future it may be desirable to use a content distribution service, such as CoDeeN to distribute node state.

3.5 Node Directory Service

In addition to being the authority on principal identities, PLC also maintains a directory of nodes and assigns each a *node_id* value that is independent of any transport addresses. Each node is assigned an effectively random 128 bit opaque value. This namespace strategy is chosen for ease of administration.

For each node_id, the PLC node directory will store a host of addressing information, including a list of IP addresses, a globally unique text name (e.g., 'node05'), a site name, GPS information, an organization name, network connectivity info, a country code, etc. Queries against this directory are made available based on all fields, allowing for human-friendly specification of nodes. Changes to info can be uploaded by a node during synchronization.

This directory addresses problems arising from nodes having dynamic IP addresses or being multi-homed. It is expected that services will be created to route based on

node_id information. This will address problems with nodes behind NATs, if the routing service is clever enough to create outbound tunnels from the NATed region.

3.6 Virtual Machine Management

Virtual machine management and resource allocation will be based on vservers [6] and SILK. Each node will run a node manager which performs privileged VM management and resource allocation operations on behalf on higher-layer resource management systems. Requests to the node manager will be driven by a a per-node PLC resource manager. Each PLC resource manager will make requests to the node manager based on a file periodically downloaded from PlanetLab central. Included in this file is a list of slivers that should exist on the node, their proportional-share resource allocations, and policy information.

3.7 Node Manager Interaction

Each PLC resource manager will periodically obtain a capability from the node manager for a course-grain resource allocation in space and in time. Using the information from PlanetLab central, it will then carve this capability up into multiple capabilities based on the desired hierarchical proportional-share resource allocations. These capabilities are then bound to slices, which may need to be dynamically created if they did not already exist.

3.8 Slice Accessibility

As before, access to slices is still done using SSH. Since PlanetLab central knows about all slices in this scheme, we will continue to rely on the existing SSH key distribution mechanism to distribute and assign SSH keys to slices. However, unlike the existing system, each slice runs a private `sshd`, and a slice bootstrap script queries the local PLC slice to determine which keys are assigned to the slice. This allows the PLC service to use an unprivileged vserver.

To simplify slice access for users, the web interface will provide a summary of access information for each slice. This will include information on which ports on each node host the SSH daemon. To further simplify this process PLC will also provide a user tool (a wrapper around SSH) to hide these complexities from the user entirely and allow access based on slice names and human-friendly information stored in the node directory.

3.9 Motivation

The centralized resource allocation scheme is simple and provides five key benefits over the existing resource allocation scheme on PlanetLab. First, it provides a rudimentary form of dynamic slice creation. Second, it provides a means for implementing policy for hierarchical, proportional-share resource allocations in a delegated manner. Third, it provides PlanetLab administrators control over resource usage at the node, slice, and principal level. Fourth, it allows the underlying resource capability architecture to be exercised in a production setting. Finally, it provides a graceful transition path for introducing new functionality based on the current infrastructure. We believe that these benefits combined with the simplicity of the proposed design outweigh the set of features not provided by this scheme. Such features include fully decentralized control, free trading of resource capabilities, and so forth. We envision that competing/experimental resource management systems will emerge to address these needs. In the meantime, the centralized allocation scheme will provide useful service and provide a concrete point of reference for competing systems in the future.

3.10 Expected Evolution

The PLC system, as described above is intended to provide a simple but usable interface to the new node manager functionality and resulting dynamic slice architecture. Once this platform has been successfully deployed, the PLC system will be improved in a number of ways.

3.10.1 Resource Reservations

The initial system provided reservations only for basic virtual machine state. Providing access to the full expressiveness of the node manager rspec datastructure will require additional mechanisms for initial resource distribution.

Once possible mechanism is to create a virtual currency that can be used to purchase reservations. The difficultly with this solution is determining appropriate pricing for resources. An alternative solution is to distribute all resource capabilities based on the current distribution algorithm, and allow users to trade them to procure desired slice configurations. Both of these approaches are complex.

3.10.2 Resource Capability Trading Support

To aid in deployment of resource markets, the central bank may provide trusted third party transaction support. If two parties wish to exchange resource capabilities for credits, the bank can participate in the transaction to prevent fraud. It can do this by being trusted by both parties to hold both the credits and the capabilities, contact the corresponding node managers to reissue the capabilities, and then complete the transaction. This would require the PLC making the node manager rcap data available to users during slice creation.

3.10.3 Efficient Secure Control Plane Overlay

To facilitate communication between virtual machines in the PLC slice, the service will create persistent SSH/TCP tunnels for XML-RPC request forwarding. For robustness this tunnel topology may be redundant. This infrastructure would allow for simple communication with nodes behind NAT domains.

Once this infrastructure is in place, the PLC service could allow other services to utilize it. The PLC can allow other slices to forward XML-RPC requests and to register their own XML-RPC servers. Other services may choose to use PLC for this for efficiency reasons, both because communication connection setup time is much lower to use the already established PLC tunnels, and because other resource such as port numbers and TCP state can be leveraged.

3.10.4 Distributed Database

Having a centralized management system has poor failure properties and is limited in scalability. Ultimately we expect that the PLC services becomes a truly distributed service, existing only within a slice and not using outside resources.

4 Example: SHARP

A second example of a resource management system that lives on the baseline architecture is SHARP [3]. SHARP is a distributed resource management architecture that allows agents, which represent sites, to exchange computational resources in a secure, fully decentralized fashion. The cornerstone of SHARP is a secure architecture for representing, validating, and delegating cryptographically protected

resource claims across a network of resource managers. SHARP also introduces mechanisms for controlled, accountable oversubscription of resource rights as fundamental tool for dependable, efficient resource management. In SHARP, agents engage in pairwise resource peering as the basis for a secure, decentralized bartering economy [1]. Resources allocated by SHARP are discovered using an external resource discovery system (e.g., Sophia [9], PIER [4], etc.). Resources are then obtained through pairwise bartering using a secure resource routing protocol.

4.1 Principals and Accountability

Principals in SHARP will be identified using X.509 certificate chains and authenticated using the SSL protocol. Principals will trust one or more certificate authorities (CAs), each of which is the root of a public key hierarchy. Initially, PlanetLab central will be the sole trusted CA and the resulting public key hierarchy will have the same structure as the one used in the centralized resource allocation scheme (Section 3.1). There, PlanetLab central asserts the identity of principles investigators and principle investigators, in turn, assert the identity of students and assume responsibility for their actions. In the centralized scheme, asserting identity and assuming responsibility translated into authenticating with the PlanetLab web site and performing associated actions. Here, analogous actions are performed in a decentralized fashion with principals issuing X.509 certificates for principals whose identities they assert and actions they are taking responsibility for.

Accountability for improper resource use will be achieved through chains of responsibility and logging of every privileged action. In our implementation, chains of responsibility will be X.509 certificate chains. The interpretation of a principal X issuing an X.509 certificate for a principal Y is that (i) X asserts that Y's true identity and contact information is the information in the certificate and (ii) that X is assuming responsibility for Y's actions on PlanetLab. A public key hierarchy represents a hierarchy of these relationships. In a SHARP-based system, principals will perform privileged actions (e.g., creating a slice) by first authenticating themselves with the SSL protocol. For every privileged action performed, a timestamped log entry will be created with the associated principal's X.509 certificate chain and the action performed.

A concrete example of how identity and accountability will interact in practice is dynamic slice creation. In SHARP, resources are obtained in a completely decentralized fashion and bound to dynamically created slices. Since this requires no coordination with PlanetLab central, extra mechanisms are needed in order to map a misbehaving sliver of a particular slice back to a responsible principal's identity

and contact information. All resource managers that live on top of the baseline capability system must be able to perform this mapping. In the centralized scheme, this mapping is straightforward since knowledge of slice to principal mappings are stored on PlanetLab central, which can be queried. In the SHARP case, there is no central entity to be queried. However, the necessary information is still readily available since timestamped logs are kept of all privileged actions performed by associated principals. In this instance, mapping a sliver back to a principal is achieved by simply reading a log file.

4.2 Secure Resource Peering

Agents engage in peering relationships by periodically exchanging resource claims. A claim is an unforgeable assertion that grants the holder of the claim access to specific resources (the resource set) over a specific time interval (the term). In SHARP, agents peer by exchanging tickets, soft claims that suggest but do not guarantee resource ownership. Tickets can then be redeemed for leases, hard claims over concrete resources that are valid for the term of the lease unless a failure occurs. In the proposed implementation, two types of tickets and leases will be supported. The first type corresponds to hierarchical, proportional-share resource allocations; the second type corresponds to reservations on hard resource allocations. Both tickets and leases are self-certifying, self-describing delegations of resource privilege in space and time and can be delegated arbitrarily between principles.

Tickets and leases can be implemented over capabilities in a relatively straightforward manner. Here, SHARP resource managers running on each node periodically will obtain a capability for a course-grain resource allocation in space and time. To support exchange of both proportional-share and hard resources in SHARP, a capability for each resource type would be obtained from the node manager. These capabilities are then passed to the site's SHARP agent which then issues tickets against these capabilities to peer with its neighbor agents. To redeem a ticket for a lease, a principal authenticates itself with the appropriate SHARP agent and presents the ticket. Upon successfully validating the ticket, the SHARP agent then communicates with a SHARP node manager to carve off a concrete resource allocation by taking an unused capability and splitting into two capabilities, the first corresponding to the resource allocation for the lease, the second corresponding to the rest of the original capability's allocation. The first capability would later be bound to a target slice.

To make the previous discussion concrete, suppose we have an agent X that peers with ten other agents: $Y_1, Y_2, \ldots, Y_9, Y_{10}$. Suppose that each agent represents a sin-

gle node site, that all resources are dedicated to the hard allocation pool, and that all resources are managed by SHARP. In this case, each SHARP resource manager on each node would periodically obtain a capability for 100% of the node's resources (for some large time interval) from the the node manager. This capability would then be passed to the site's SHARP agent, which would issue tickets against this capability to its peers. For example, X might partition its resources evenly across all its peers and issue 10 tickets, each for 10% of its resources, to $Y_1, Y_2, \ldots, Y_9, Y_{10}$. Y_i obtains a lease at X by presenting its ticket to X. X, having one big unused capability for 100% of its resources, would then split that capability into two capabilities, a 10% capability to back the lease and a 90% capability to handle future ticket redemptions. The 10% capability would subsequently get bound to a target slice.

SHARP makes a distinction between tickets and leases to allow for oversubscription. Oversubscription allows an agent to issue more tickets than it has physical resources for. Oversubscribed claims can benefit resource utilization by statistical multiplexing, and support replicated tickets to limit the damage from resource loss if an agent fails or becomes unavailable. For tickets corresponding to hard resource allocations, oversubscription is implemented by issuing more tickets than there are physical resources and by rejecting claims when conflicts occur. For tickets corresponding to proportional-share resource allocations, oversubscription for utilization is implemented by simply issuing a fixed set of tickets for the proportional-share pool and letting the underlying proportional-share scheduler ensure that the resources are fully utilized. As with the hard allocation case, the SHARP resource manager must detect duplicate use of the same ticket resource sets (e.g., redeeming the same ticket twice).

4.3 Secure Resource Routing Protocol (SRRP)

Resources in SHARP are obtained through a two-phase process. First, resources of interest are found using an external resource discovery service. Second, pairwise bartering is done for desired resources at target sites. For resource discovery, we assume the existence of a resource discovery system which accepts resource queries as input and returns query results as output. In the case of SHARP, query results include both the set of nodes matching the resource query as well as addresses for associated agents that are responsible for issuing tickets for those nodes. Once target agents have been identified, pairwise bartering is done to each agent to obtain tickets for the desired resources. For each target agent, there could be multiple bartering paths from the source trying to acquire the resources and the destination

agent that has the desired resources. Each path consists of a sequence of directed edges, each of which specifies both an exchange rate and a capacity.

SHARP uses a Secure Resource Routing Protocol (SRRP) to both announce bartering paths and to securely route requests for tickets along bartering paths. Path information is disseminated using a secure link-state routing protocol and used to construct a routing table at each agent which stores lowest cost bartering paths to each destination agent. We define cost as the the number of tickets the source has to relinquish in order to obtain tickets at the destination agent. We define capacity on each edge (X,Y) as the number of Y tickets that can be obtained through X. We expect that routing based on lowest cost paths are likely to be the common case. Lowest cost routing with simple constraints are also likely to be common. Routing with constraints is desirable for using cheap routes that avoid specific agents that are faulty, malicious, or unresponsive.

Pairwise bartering along a bartering path is done using source-based routing to allow agents to route bartering requests along paths of least cost. Pairwise bartering along a path is done using the tickets obtained as a result of peering relationships. For example, if A peers with B, B peers with C, C peers with D, ..., and Y peers Z, A obtains tickets from Z through a sequence of pairwise bartering towards Z and a sequence of delegations back to A. Towards Z, A hands B a set of B tickets, B hands C a set of C tickets, and so on. The end-to-end bartering exchange rate from A to Z is the product of the pairwise exchange rates. Once the Z tickets have been obtained, the Z tickets are then delegated back to A along the reverse path. Once obtained, these delegated Z tickets can then be directly redeemed by A to obtain a lease.

A key benefit of SHARP's approach to bartering is that it allows resource exchange between arbitrary agents while avoiding $O(n^2)$ exchange rates, a well-known limitation in bartering economies. It achieves this through a combination of pairwise resource peering and sequences of pairwise bartering along minimum-cost routes from source agents wanting resources to destination agents that have desired resources. The implication of this is that the system has much better scaling properties. It also significantly lowers the barrier to entry for new sites wanting to contribute and share resources in PlanetLab. New sites only have to reason about exchange rates with a (potentially small) subset of sites in PlanetLab to begin participating in the bartering economy.

SRRP uses a combination of cryptographically-signed receipts and witnesses (i.e., similar to mix-net routing in Free Haven [2]) to handle cases where pairwise bartering fails due to faulty, malicious, or unresponsive agents. Through source-based routing, the source already has some control over the path a request is routed over.

However, making this work in practice requires that each agent on the path is both available and behaving properly. In the event that an agent along the path does not have these properties, we would like to be able to identify at which point along the route the routing request failed and to have some indication as to why it failed. For example, when routing from A to Z, suppose G drops the request and simply uses the tickets it obtained from F to barter for its own advantage. By requiring timestamped, signed receipts accepting responsibility at each hop, F could present a copy of this receipt to A, which might subsequently avoid G in the future as a result. Having observed G's behavior, F may also wish to avoid routing through G in the future. Given persistent bad behavior from G, F might even stop peering with G altogether.

Initially, tickets used for peering and routing will represent shares of resources at a particular site. Each agent will issue tickets for a set of nodes and represent aggregate node resources as a set of shares. These shares will subsequently be packaged up as tickets and exchanged as part of peering. For example, an agent issuing tickets for three nodes might represent these nodes as 300 shares with each node's resources contributing 100 shares and where each share corresponds to 1% of a node's CPU, memory, network, and disk resources. A key benefit of viewing shares this way is that exchange rates can then be represented as scalars and standard graph algorithms can be used to compute optimal paths. A potential disadvantage with this approach is resource fragmentation since resource requests must scale up across all resources. Redeeming tickets for leases on specific nodes is done by sending the ticket (representing some number of shares) and a mapping of the ticket's shares to a set of nodes. It is up to the requester to pick an appropriate mapping based on resource discovery information.

4.4 Dynamic Slice Creation

Dynamic slice creation in SHARP is achieved by obtaining leases on target nodes and by binding the resources associated with those leases to a newly created slice. Tickets on target nodes will be obtained using the bartering mechanisms previously described. These tickets will then redeemed for leases, which are then sent as part of a dynamic slice creation request to the nodes that will comprise the slice. With respect to the underlying capability architecture, SHARP agents will maintain the mappings of SHARP leases to capabilities. When a lease is initially allocated, so is the capability, but it is unbound to a particular slice. Note that these leases could then be potentially resold in a market for computational resources. Only when a slice creation request is received does the capability get bound to a specific slice.

Leases can also be bound to existing slices (i.e., to implement lease renewal for a long-lived slice).

Authorization to create slices could be implemented using additional certificates. We have implemented a rudimentary trust management system that allows certificates which delegate arbitrary privilege to be expressed and verified based on a security policy. As an alternative, we could instead always allow principals to create slices (modulo limits on the node) as long as there is a chain of responsibility. The upside of this approach is that it avoids extra certificates. The downside is that it makes it harder to express certain types of policy (e.g., only PIs and their immediate underlings can create slices). As with the centralized scheme, we rely on slice names that are tuple of a principal_id and an XXX bit slice_tag. Here, the principal_id might be the principal ID of the SHARP resource management system and the XXX bit slice_tag might be constructed using the SHA1 hash of the requesting principal's public key and a name in that principal's local slice namespace. Principals would authenticate with the SHARP resource manager to prove they have the private key that allows use of their part of the slice namespace.

4.5 Administrative Policies

As with the centralized scheme, administrators at PlanetLab central will be able to set policies on nodes, slices, and principals that apply to resources allocated by all resource management systems, including SHARP. Policies on nodes and slices will be implemented as before. Policies on principals will be slightly different, depending on the implementation. One possibility is that we require the public key hierarchy to be known and apply policy that way. Another is that we require a completely separate policy infrastructure for principals based on PKI. The advantage of the first approach is that policy could still be centrally applied to principals at any level of the PKI hierarchy. One possible approach here would be to have a service that allows certificates to be registered and to require that all certificates on a chain be registered before authenticating a principal. A simple example of such a service would be a web-based registration service (e.g., using the existing web accounts) combined with centralized distribution of a list of SHA1 hashes of all registered certificates.

4.6 VM Management and Resource Allocation

Virtual machine management and resource allocation will be based on vservers and SILK. Each node will run a node manager that performs privileged VM man-

agement and resource allocation on behalf of higher-layer resource management systems. Requests to the node manager will be driven by a per-node SHARP resource manager and a per-site SHARP agent on each node. Virtual machine management will be similar to the PlanetLab central approach, the key difference being that decisions to create, delete, and manage slices will be based on requests from service managers (e.g., run by individual users), as opposed to a file downloaded from PlanetLab central. SSH public keys will be installed as part of dynamic slice creation by sending the set of authorized keys to SHARP resource managers for all nodes in the slice. SHARP resource managers can then be queried by node managers when running slice bootstrap scripts to set up SSH access.

5 Implementation

The implementation will consist of several pieces whose implementation can proceed largely in parallel. Working bottom-up, these pieces include:

- Hierarchical proportional-share scheduling layer (acb)
- Node manager and resource capabilities layer (tspalink)
- Resource specification language for prop-share/reservations (scott)
- PlanetLab central resource management system (tspalink)
- SHARP resource management system (bnc, fu)

Note that the assignment of names to the different pieces above is just an approximation.

5.1 Schedule

The implementation schedule is shown in Table 1.

Contributors

This document includes contributions by Andy Bavier, Mic Bowman, Paul Brett, Yun Fu, Larry Peterson, Timothy Roscoe, Amin Vahdat, and Mike Wawrzoniak. This document was derived from PDN-02-005.

Date	Milestone
Jul 18	Draft of this document. Send out for feedback.
Jul 25	Interfaces for components, simple resource specification language.
Aug 1	Simple tests written for VM mgnt, sched, nodemgr, capabilities.
Aug 4	Node manager and resource capabilities layer.
Aug 4	Hierarchical prop-share CPU and network bandwidth.
Aug 4	SHARP prototype (no integration w/ node manager).
Aug 11	Refine resource specification language.
Aug 11	PlanetLab central resource allocation (dynamic slice creation + keys).
Aug 11	SHARP prototype (integration w/ node manager, testing).
Aug 18	PlanetLab central resource allocation (add hierarchical prop-share).
Aug 18	Per-slice TCP/UDP ports and IP tables.
Aug 25	PlanetLab central resource allocation (add node/slice/principal policies).
Aug 25	SHARP prototype (optimizations, full decentralization).
Sep 1	Testing and bug fixes.
Sep 8	Deploy PlanetLab central resource allocation.

Table 1: Implementation Schedule.

planetlab-slices@lists.sourceforge.net.

References

[1] B. Chun, Y. Fu, and A. Vahdat. Bootstrapping a distributed computational economy with peer-to-peer bartering. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[2] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000. Updated December 2000.

[3] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. Sharp: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003. To appear.

[4] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *Proceedings of the 1st International Workshop on Peer-to-peer Systems*, March 2002.

[5] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[6] Solucorp. Virtual private servers and security contexts (`http://www.solucorp.qc.ca/miscprj/s_context.hc`), 2003.

[7] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, 1994.

[8] C. A. Waldspurger and W. E. Weihl. An object-oriented framework for modular resource management. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, pages 138–143, October 1996.

[9] M. Wawrzoniak. Sophia knowledge plane (`http://www.cs.princeton.edu/~mhw/sophia`), 2003.