

# ***Slice-Based Facility Architecture***

*Larry Peterson, Princeton*

*Soner Sevinc, Princeton*

*Jay Lepreau, Utah*

*Robert Ricci, Utah*

*John Wroclawski, USC/ISI*

*Ted Faber, USC/ISI*

*Stephen Schwab, Sparta*

*Scott Baker, Arizona*

Draft Version 1.04

April 7, 2009

This work is supported in part by NSF grants CNS-0540815 and CNS-0631422.

## **Table of Contents**

1	Introduction.....	3
2	Principals .....	3
3	Abstractions.....	4
3.1	Components.....	4
3.2	Slices.....	5
4	Names & Identifiers .....	6
4.1	Registries .....	6
5	Data Types.....	8
5.1	RSpec.....	8
5.2	Registry Record .....	9
5.3	Ticket.....	10
5.4	Credentials .....	10
6	Interfaces.....	11
6.1	Registry Interface .....	12
6.2	Slice Interface .....	13
6.2.1	Instantiating a Slice .....	13
6.2.2	Provisioning a Slice .....	14
6.2.3	Controlling a Slice .....	15
6.2.4	Slice Information.....	15
6.3	Component Management Interface.....	16
7	Authorization and Access Control.....	17

## 1 Introduction

This document defines the minimal set of interfaces and data types that permit a federation of slice-based network substrates to interoperate. The specification is designed to support federation among facilities like PlanetLab, Emulab, VINI, and GENI – and assumes the reader is familiar with those systems – but is intended to support a much broader range of designs than those systems embody.

Although this effort grew out the GENI Initiative, it does not currently have any official standing with GENI. Lacking any such sponsorship – and hoping to foster much broader acceptance – we refer to the architecture defined in this document as the *Slice-based Facility Architecture* (SFA).

## 2 Principals

The SFA recognizes four key actors:

- *Owners* of parts of the network substrate, who are therefore responsible for the externally visible behavior of their equipment, and who establish the high-level policies for how their portion of the substrate is utilized.
- *Operators* of parts of the network substrate, often working for owners, whose job it is to keep the platform running, provide a service to researchers, and prevent malicious or otherwise damaging activity exploiting the platform.
- *Researchers* (and *developers*) employing the network substrate, for running experiments, deploying experimental services, measuring aspects of the platform, and so on.
- *Principle Investigators (PI)* representing research organizations that take responsibility for individual researchers at their site.

The SFA must mediate the following activities:

- Allow owners to declare resource allocation and usage policies for substrate facilities under their control, and to provide mechanisms for enforcing those policies. The assumption is that there will be multiple owners and it will be a *federation* of these facilities that will form the entirety of the network.
- Allow operators to manage the network substrate, which includes installing new physical plant and retiring old or faulty plant, installing and updating system software, and monitoring the network for performance, functionality, and security. Management is likely to be decentralized: there will be more than one organization administering disjoint collections of sites.
- Allow researchers to create and populate slices, allocate resources to them, and run experiment-specific software in them. Some of this functionality, such as convenient installation of software, including libraries or language runtimes, may be provided by higher-level services; the SFA aims to support the deployment and configuration of such services.

- Allow PIs to identify the set of researchers at their organization that be permitted to utilize the facility.

To this end, the SFA defines three principals:

- A *management authority* (MA) is responsible for some subset of substrate components: providing operational stability for those components, ensuring the components behave according to acceptable use policies, and executing the resource allocation wishes of the component owner.
- A *slice authority* (SA) is responsible for the behavior of a set of slices, vouching for the researchers running experiments in each slice and taking appropriate action should the slice misbehave.
- A *user* is a person playing one or more roles in a facility – a researcher that wishes to run an experiment or service in a slice, an operator that manages some part of the substrate, a PI at an institution that conducts research on the facility, or an owner that contributes resources to a facility.

Note that we expect there to be *end-users* (or *clients*) of the services deployed in slices, but this report offers no guidance on how these individuals interact with the system, as this is a slice-specific concern.

## 3 Abstractions

The SFA defines two key abstractions: *components* and *slices*.

### 3.1 Components

*Components* are the primary building block of the architecture. For example, a component might correspond to an edge computer, a customizable router, or a programmable access point.

A component encapsulates a collection of *resources*, including physical resources (e.g., CPU, memory, disk, bandwidth) logical resources (e.g., file descriptors, port numbers), and synthetic resources (e.g., packet forwarding fast paths). These resources can be contained in a single physical device or distributed across a set of devices, depending on the nature of the component. A given resource can belong to at most one component.

Each component is controlled via a *component manager* (CM), which exports a well-defined, remotely accessible interface. The component manager defines the operations available to user-level services to manage the allocation of component resources to different users and their experiments. Typically, a component's CM runs on the component itself, although a remote *proxy CM* can control components that are unable to host a CM.

A management authority (representing the wishes of the owner) establishes policies about how the component's resources are assigned to users.

It must be possible to multiplex (slice) component resources among multiple users. This can be done by a combination of virtualizing the component (where each user acquires a virtual copy of the component's resources), or by partitioning the component into distinct resource sets (where each user acquires a physical partition of the component's resources). In both cases, we

say the user is granted a *sliver* of the component. Each component must include hardware or software mechanisms that isolate slivers from each other, making it appropriate to view a sliver as a “resource container.”

A sliver that includes resources capable of loading and executing user-provided programs can also be viewed as supporting an *execution environment*. Slivers that support such execution environments are said to be *active slivers*. Other (non-active) slivers might correspond to communication resources; e.g., a tunnel, VLAN, circuit, or light-path.

Sometimes it is convenient to represent a collection of components as a single *aggregate*. For example, one might treat all the nodes and links in backbone network as an aggregate. Such an aggregate can be accessed via an *aggregate manager* (AM), which exports the same interface as an individual component.

## 3.2 Slices

From a researcher's perspective, a *slice* is a substrate-wide network of computing and communication resources capable of running an experiment or a wide-area network service. From an operator's perspective, slices are the primary abstraction for accounting and accountability – resources are acquired and consumed by slices, and external program behavior is traceable to a slice, respectively.

A slice is defined by a set of slivers spanning a set of network components, plus an associated set of users that are allowed to access those slivers for the purpose of running an experiment on the substrate. That is, a slice has a name, which is bound to a set of users associated with the slice and a (possibly empty) set of slivers.

There are three unique stages in the lifetime of a slice, each corresponding to an action (operation) that can be performed on a slice:

- **Register:** the slice exists in name only and is bound to a set of users;
- **Instantiate:** the slice is instantiated on a set of components and resources assigned to it;
- **Activate:** the slice is activated (booted), at which point it runs code on behalf of a user.

A slice has to be registered and bound to a set of users before it can be instantiated, and it must be instantiated before being it can run code or be accessed by a user.

Slices are registered in the context of a *slice authority* – a principal that takes responsibility for the behavior of the slice. A slice is registered only once, but the set of users bound to it can change over time. A slice registration has a finite lifetime; the responsible slice authority must refresh this registration periodically.

Instantiating a slice effectively configures the slice on a set of components; this step can be repeated multiple times. In fact, instantiating often involves two sub-steps: a slice is first instantiated on a set of components with only best-effort resources assigned to it, and later provisioned with additional (perhaps guaranteed) resources, for example, for the duration of a single experiment.

An experiment or service then “runs in” a slice. Multiple experiments can be run in a single slice. For each run, the experiment may change parameters but leave the slice configuration (instantiation) unchanged, or it may change either the set of components or the resources assigned on those components, or both. How rapidly a slice can be reconfigured to support a new experiment depends on the implementation of the instantiation and provisioning operations.

## 4 Names & Identifiers

The SFA defines *global identifiers* (GID) for the set of objects that make up the federated system, which include components, slices, services, and the various principals described in Section 2. In short, every entity in the system that wishes to communicate has a GID.

GIDs form the basis for a correct and secure system, such that an entity that possesses a GID is able to confirm that the GID was issued in accordance with the SFA and has not been forged, and to authenticate that the object claiming to correspond to the GID is the one to which the GID was actually issued.

Specifically, a GID is a certificate that binds together three pieces of information:

GID = (PublicKey UUID, Lifetime)

The object identified by the GID holds the private key corresponding to the PublicKey in the GID, thereby forming the basis for authentication. The UUID is a Universally Unique Identifier [X667] for the object. An object’s UUID is immutable (it stays the same if the PublicKey changes) and absolute (it identifies the same object throughout the entire system). The Lifetime says how long the GID is valid; GIDs need to be “refreshed” periodically. An authority is identified by its own GID, hence, any entity may verify a GID via cryptographic keys that lead back, possibly in a chain, to a well-known root or roots.

When necessary for clarity, we distinguish between the *plain* GID denoting an object (the 3-tuple given above), the *signed* GID (the above 3-tuple plus a signature generated by a responsible authority), and the *bundled* GID (the set of signed GIDs, sufficient to verify the GID back to a trusted root authority). Note that the signed GID is, in fact, a certificate.

This design reflects three engineering decisions. First, one could use the PublicKey rather than the UUID to uniquely identify each object, but this would imply that the unique key for each object change whenever the key changes (e.g., if the corresponding private key is ever compromised). The expectation is that the UUID is an immutable object identifier. Second, it is possible for an authority to forge the UUID it assigns to an object. The UUID can include one or more sub-strings (i.e., prefixes) that uniquely identify the authorities that signed the certificate – making it possible to verify that the UUID has not been forged – but ultimately one has to recognize when a given authority cannot be trusted to produce valid UUIDs. Third, multiple authorities can sign (accept responsibility for) the same GID, in which case the GID would be bound to more than one name (as described next).

### 4.1 Registries

A *registry* maps *human-readable names* (HRN) to GIDs, as well as records others domain-specific information about the corresponding object, such as the URI at which the object’s manager can

be reached, an IP or hardware address for the machine on which the object is implemented, the name and postal address of the organization that hosts the object, and so on.

An HRN for an object identifies the sequence of authorities that are responsible for (have vouched for) the object. While the SFA allows for an arbitrary organization of registries, for simplicity of exposition, this document focuses on a hierarchical name space corresponding to a hierarchy of authorities that have delegated the right to create and name component and slice objects. This hierarchy assumes a top-level naming authority trusted by all entities, resulting in names of the form:

`top-level_authority.sub_authority.sub_authority.name`

For example, “geni” and “planetlab” might be top-level authorities;<sup>1</sup> it is possible that other similar authorities might federate in accordance with the SFA. This is not to imply that all federation is strictly among top-level authorities, since even in the context of a single top-level authority, we allow for multiple autonomous MAs that agree to federate their resources.

The registry maintains information about a hierarchy of *management authorities*, along with the set of components for which the MAs are responsible. It binds a human-readable name for components and MAs to a GID, along with a record of information that includes the URI at which the component’s manager can be accessed, other attributes that might commonly be associated with a component (e.g., hardware addresses, IP addresses, DNS names), and contact information for the users (owners and operators) responsible for those components. For example,

`geni.us.backbone.nyc`

might name a component at the NYC PoP of GENI’s US backbone. In this case, the `geni.us.backbone` management authority is responsible for the operational stability of the set of components in the backbone network.

The registry also maintains information about a hierarchy of *slice authorities*, along with the set of slices for which the SAs have taken responsibility. It binds a human-readable name for slices and SAs to a GID, along with a record of information that includes contact information for the set of users (PIs and researchers) responsible for those slices. For example,

`planetlab.eu.inria.dali`

might name a slice created by the PlanetLab slice authority, which has delegated to the EU, and then to INRIA, the right to approve slices for individual projects (experiments), such as Dali. PlanetLab defines a set of expectations for all slices it approves, and directly or indirectly vets the users assigned to those slices.

Note that both the GENI and PlanetLab management authorities are expected to maintain an operational set of components capable of hosting experiments, and their respective slice authorities are expected to approve slice creation on behalf of network and distributed systems

---

<sup>1</sup> The GENI literature refers to a Clearinghouse, which can be viewed as “trust anchor.” A top-level authority (e.g., PlanetLab) is an example of such a trust anchor.

researchers. Because it is possible that other related facilities will federate with GENI and PlanetLab, and there will be other uses of the greater federated system, we allow for the possibility that other top-level slice authorities may support other policies and purposes. For example, there could exist a top-level slice authority that permits slices running for-profit services.

More generally, this document's focus on a global hierarchy should not be taken to imply that all authorities are known to a handful of globally trusted roots. For example, a consortium of organizations might agree to create (and subsequently trust) a collection of sub-authorities, slices, and users without being known globally; e.g.,

our\_private\_consortium.my\_organization.some\_slice

There could even be stand-alone authorities that, if someone was willing to trust them, could participate in an SFA-based facility.

Note that human-readable names are useful because they are easy for humans to remember and state, which makes them particularly important in crafting policy statements. For example, an owner might specify a policy that says a component is willing to allocate up to X% of its capacity to slices belonging to the `planetlab.eu.inria` authority, but no more than Y% of its capacity to the specific slice `geni.bbn.p2p`.

Finally, note that a registry may be distributed, where a server that implements one portion of a hierarchy includes a pointer (URI) to a server that implements a sub-tree of the hierarchy. When necessary for clarity, we distinguish between the *global registry* (the entire collection of registry information), an *authority registry* (one level of the global registry corresponding to the information maintained by a single slice or management authority), and a *registry server* (a network-accessible server process that implements some sub-tree of the global registry, including one or more authority registries).

## 5 Data Types

The SFA defines four key data types in addition to GIDs. This section defines these data types at an abstract level. A candidate set of concrete representations is defined elsewhere. This section also identifies potentially useful library routines that can be used to manipulate these data types, but these routines are also defined elsewhere.

### 5.1 RSpec

A *resource specification* (RSpec) describes a component in terms of the resources it possesses and constraints and dependencies on the allocation of those resources. The exact form of an RSpec is still being defined elsewhere, but in addition to information about component resources, each RSpec includes the following two fields:

(StartTime, Duration)

indicating the period of time for which the requested resources are desired (or granted resources are available). By default, `StartTime=Now` and `Duration=Indefinite`.



**Note:** An RSpec might also include a “feedback URI” that the component uses to notify the slice when an allocation is about to change underneath it.

## 5.2 Registry Record

A registry records facts about the objects in the system (e.g., components and slices), and the principals (e.g., users, MAs and SAs) that use and authorize them. Registry records are defined to be of the following form:

Record = (HRN, GID, Type, Info)

Where HRN and GID are as defined in Section 4,

Type = SA | MA | Component | Slice | User

and

Info = (PI[ ], Organization), if Type = SA

Info = (Owner[ ], Operator[ ], Organization), if Type = MA

Info = (URI, LatLong, IP, DNS), if Type = Component

Info = (URI, Researcher[ ], InitScript), if Type = Slice

Info = (PostalAddr, Phone, Email, AuthTokens[ ]), if Type = User

When present, the URI field references an object manager that exports one or more of the standard SFA interfaces. For example, a component record might point to a Component Manager that implements the Slice and Management interfaces defined in 6.2 and 6.3, respectively, while a slice record might point to an agent that assists users in creating and controlling their slices, although users are allowed to implement this functionality without the assistance of some external agent. We sometimes call such an agent a *slice manger*.

The SA, MA, and Slice record types include references to (GIDs for) one or more User records. They are denoted PI, Owner, Operator, and Researcher, respectively. These labels signify the role the user(s) affiliated with that entity plays, but these labels are descriptive only. What really matters is the set of rights encoded in the credentials granted to various users, as described in Section 5.4, and further elaborated in Section 7.

The InitScript field in a Slice record stores a minimal initialization script that executes when a sliver is instantiated on a component. For example, it might fetch and execute a larger boot program from some URI. As another example, it might install a public key that can subsequently be used by a remote agent (e.g., slice manager) to securely access and initialize the sliver. Note that this implies all active components be able to interpret a common, but minimal, scripting language.

The AuthTokens field in a User record stores the authentication tokens needed to access slivers created on behalf of the corresponding user. We expect different types of components will support different access methods (e.g., ssh) for slivers they host, with the related tokens recorded here. We leave the issue of how AuthTokens are distributed to components that host a given slice (and subsequently updated when news users are bound to the slice) as an implementation issue. These tokens are stored in the registry, but responsibility for

distributing/updating these tokens falls to either the slice manager that created the slice or the component that hosts the slice. It is not the responsibility of the registry.

Note that we expect the information available in a registry to be relatively static. To learn more detailed and dynamic information about a component, for example, one needs to call the component directly using the URI for the Component Manager identified by the registry. The interface exported by a CM includes operations for leaning the resources available on that component.

Also note that a registry may contain multiple records with the same HRN, each of a different type. For example, `planetlab.princeton` might name a slice authority (have an SA record), a management authority (have an MA record), and a component aggregate (have a Component record). Each of these different cases would correspond to a distinct object, and hence, have a unique GID. (In practice, however, each such GID could share the same public key.)

Finally, we expect additional record types will be added to the registry over time. For example, the registry might record information about various user-level services, some of which may run in a slice (e.g., a software distribution service itself runs in a slice of the network substrate) and some of which run on a service outside the substrate (e.g., a slice manager that exports a GUI for specifying and instantiating slices.) Such services will then be treated as first-class objects in system, complete with their own GID.

### 5.3 Ticket

A component signs an RSpec to produce a *ticket*, indicating a promise by the component to bind resources to the ticket-holder at some point in time. Such tickets are “issued” by a component, and later “redeemed” to acquire resources on the component. Tickets may also be “split,” effectively passing resources from one principal to another.

The SFA defines the tickets to includes the following information:

$\text{Ticket} = (\text{RSpec}, \text{GID}, \text{SeqNum})$

where RSpec describes the resources for which rights are being granted by the component; GID identifies the slice or slice authority to which rights to allocate the resources are being granted; and the SeqNum ensures that the ticket is unique. This information is signed by the component that issues the ticket.

### 5.4 Credentials

A credential carries the rights issued to a particular principal. For example, a user might be granted credentials that allow it to instantiate a slice in a set of willing components for the period of time during which the slice is said to be live. A credential is given by the 6-tuple:

$\text{Credential} = (\text{CallerGID}, \text{ObjectGID}, \text{ObjectHRN}, \text{Expires}, \text{Privileges}, \text{Delegate})$

where CallerGID identifies the principal to which the credential has been issued; ObjectGID and ObjectHRN identify the object for which the credential applies; Expires says how long the credential is valid; Privileges identifies the class of operations the holder is allowed to invoke; and Delegate indicates whether the holder is permitted to delegate the credential to another principal.

A credential is signed by the responsible authority, and similarly re-signed when delegated. Although not defined in this document, we assume there exists a library routine that a user calls to delegate a credential to another principal. This routine must allow the holder of a credential to delegate a subset of the privileges it holds, as well as clear the `Delegate` field so that the credential cannot be re-delegated.

Each privilege implies the right to invoke a certain set of operations on one or more of the SFA interfaces. Privileges include:

<b>Privilege</b>	<b>Interface</b>	<b>Operations</b>
authority	Registry	<i>all</i>
refresh	Registry	Remove, Update
resolve	Registry	Resolve, List, GetCredential
pi	Slice	<i>all</i>
instantiate	Slice	GetTicket, CreateSlice, DeleteSlice, UpdateSlice
bind	Slice	GetTicket, LoanResources
control	Slice	UpdateSlice, StopSlice, StartSlice, DeleteSlice
info	Slice	ListSlices, ListComponents, GetSliceResources, GetSliceBySignature
operator	Management	<i>all</i>

Section 7 defines the expected policy for generating credentials, given the information contained in the relative registry record.

## **6 Interfaces**

The following describes, in high-level terms, the interfaces provided by the core set of SFA objects. A candidate set of concrete interfaces is defined elsewhere.

Not included in the following description is a definition of the secure remote invocation mechanism that allows the caller to invoke one of the operations defined below on a specified object manager. Such a mechanism allows the caller to identify the callee with a URI, and then facilitates both sides using their respective GIDs to authenticate the other. We expect the architecture to accommodate multiple such invocation mechanisms.

## 6.1 Registry Interface

The registry interface supports the following six operations:

```
Register(Credential, Record)
Remove(Credential, Record)
Update(Credential, Record)
Record = Resolve(Credential, HRN, Type)
Record[ ] = List(Credential, HRN, Type)
Credential = GetCredential(Credential, HRN, Type)
```

The first two operations are used to register and un-register objects and principals, while the third operation is used to update information about an entry. Each record includes live-ness information (the `Lifetime` field contained in the `GID`), which must be periodically refreshed (using `Update`) or the record is automatically removed. The fourth operation is used to learn the information bound to a given `HRN` and the fifth operation is used to retrieve information about the set of objects managed by the named authority.

All operations are interpreted relative to a `Credential` that specifies the context (authority) in which the operation is applied. For example, invoking `Register` with a `Credential` that specifies `planetlab.princeton` and `Type=Slice` registers new slice with the Princeton slice authority.

The final operation allows a principal to retrieve credentials corresponding to the named object. For example, a user might invoke `GetCredential`, giving his or her user credentials as the first argument, to retrieve the credentials associated with the named slice. The `Type` argument is used to differentiate among multiple records with the same name, so for `Type=Slice`, the return value is a “slice credential” that can subsequently be passed to the operations defined in the next section. Similarly, a call to `GetCredential` with `Type=SA` returns a “registry credential” that can subsequently be used to operate on records belonging to the named authority.

Since registries return credentials, and all rights encoded in those credentials flow from a chain of authorities, one might view a registry as an agent of an authority, but this isn't necessarily the case. A registry simply stores information about objects, including credentials that can subsequently be retrieved with the `GetCredential` call. One implementation strategy is to conflate the authority and the registry, that is, to embed an authority's policy for deciding what rights to include in a credential in a registry. This simplifies the implementation, but has the disadvantage of expanding the trusted code base (TCB) to include the registry, when in fact, it is only the function that creates and signs the credential that must be trusted. An alternative implementation strategy is for each authority to isolate its credential creation function (and associated policy) in a minimal TCB, with the authority simply storing credentials in the registry where users are allowed to retrieve them.

Users typically bootstrap their “registry credentials” through an out-of-band process. For example, a researcher and a PI might jointly construct a new `GID` for the researcher (typically the researcher provides the public key and the PI provides the `UUID` and sets the lifetime for the `GID`), the researcher passes the contact information needed to complete the registry record to the PI, and the PI registers the newly constructed record (including the new user's `GID`) in

the authority registry for which it has the necessary “registry credentials.” We assume the researcher then constructs a “bootstrap credential” (using its new GID as both the CallerGID and ObjectGID) and calls `GetCredential` to retrieve the “registry credential,” which it then uses for subsequent registry calls. Alternatively, a user that already has a GID, perhaps issued by some other authority, may pass this signed GID to the PI out-of-band, and the PI is free to continue the registration process using this GID if it trusts the original signing authority.

Most of the calls defined in the next two sections take a credential as an argument. This credential, coupled with the exchange of GIDs assumed by the underlying invocation mechanism, is sufficient for the callee to determine if the caller is allowed to invoke the specified operation. Notice, however, that the validity of the credential is subject to the accuracy of the GID’s `Lifetime` field; that is, an authority can explicitly delete a GID (and associated registry record) after issuing the credential, but before its lifetime expires. A conservative callee is free to call the registry and confirm that the GID is still valid (has not been deleted). This check is functionally equivalent to checking a revocation list. The SFA does not define a distribution mechanism for such revocations, but a third party service could poll registries for records that have been explicitly deleted before the GID’s `Lifetime` has expired, implementing such a distributed revocation list.

## 6.2 Slice Interface

Once a slice has been registered with a trusted slice authority, any user bound to the slice can retrieve a credential giving it the right to invoke the following operations on a component to instantiate and provision the slice. Note that a single component is able to create only local slivers, meaning that the following operations must be invoked on each component that the slice is expected to span, perhaps indirectly through a proxy or aggregate acting on behalf of a set of components. Thus, individual components, aggregates representing sets of components, aggregates of aggregates, and proxies for components all support the slice interface.

It is important to keep in mind that the slice interface is used to create and control slices; it defines a “control plane” for slices. The consequence of invoking these operations is an instantiated slice – or more specifically, a collection of slivers distributed across the components of the network substrate – but this is where the control plane’s reach ends. The behavior of those individual slivers – that is, how they are accessed, programmed, and used – is component-specific. For example, the SFA does not define an API for “logging into” a sliver, and as noted in Section 5.2, it is an implementation detail as to how the keys used to access a sliver are actually distributed to the component hosting the sliver.

### 6.2.1 Instantiating a Slice

A combination of four operations are used to instantiate (embed) a slice:

```
Ticket = GetTicket(Credential, RSpec)
RedeemTicket(Ticket)
ReleaseTicket(Ticket)
CreateSlice(Credential, RSpec)
```

A user invokes the first operation on a component to acquire rights to component resources. The returned ticket effectively binds the slice to the right to allocate on that component the

requested resources. Whether or not the call succeeds depends on the local resources available on the component, and the resource allocation policy implemented by the component (on behalf of the component owner). The **Credential** parameter identifies the slice or slice authority requesting the resources, and indicates the period of time for which the slice's registration is valid; the component likely limits the returned ticket's duration accordingly. The **Credential** must include the **instantiate** or **bind** privilege.

Once a principal possesses a ticket, it can create a sliver on the component and bind new resources to an existing sliver by invoking the **RedeemTicket** operation. Creating a new sliver requires the **instantiate** privilege and augmenting an existing sliver with additional resources requires the **bind** privilege. The **ReleaseTicket** call undoes a ticket allocation.

Alternatively, a caller can embed a slice with a single **CreateSlice** call. This call is essentially equivalent to back-to-back **GetTicket/RedeemTicket** calls.

Note that **RedeemTicket** and **SplitTicket** (next section) are the only operations that do not take a **Credential** as an argument. Instead, both take a **Ticket**, which effectively plays the role of a credential in the sense that it says what set of resources the corresponding principal has the right to allocate or bind. A principal must have the **instantiate** or **bind** privilege to call **GetTicket**, but once a ticket exists, the principal to whom the resources are bound may call **SplitTicket**.

The **GetSliceResources** call (defined below) can be used to learn the specific resources were actually assigned to the slice.

### **6.2.2 Provisioning a Slice**

Three operations are used to manipulate the resources bound to a slice:

**NewTicket** = **SplitTicket**(**Ticket**, **GID**, **RSpec**)

**LoanResources**(**Credential**, **GID**, **RSpec**)

**UpdateSlice**(**Credential**, **RSpec**)

An entity that holds a ticket uses the first operation to split off a portion of the corresponding resources, effectively creating a new ticket. The **GID** parameter specifies the slice to which the ticket's resources are to be bound. Note that splitting a ticket requires calling the entity that originally issued the ticket, independent of how many times the ticket has previously been split. (In contrast, a credential can be delegated locally, without contacting the issuer of the credential.) This new ticket can be redeemed using the **RedeemTicket** operation (described above); resulting in either a new slice being instantiated on the component or additional resources being bound to an existing slice.

A slice uses the second operation to loan some of its current resources to the specified slice. A slice can learn its allocation on the component using the **GetSliceResources** operation (described below). Loaned resources are transferred from one slice to another without being encapsulated in a ticket.

A user invokes the third operation to request that additional resources – as specified in the **RSpec** – be allocated to the slice. Note that **UpdateSlice** and **CreateSlice** can be viewed as

alternative name for the same operation: the former creates the slice if it does not already exist, while the latter updates the slice if it already exists.

### 6.2.3 Controlling a Slice

Component managers support four control operations:

```
StopSlice(Credential)
StartSlice(Credential)
ResetSlice(Credential)
DeleteSlice(Credential)
```

where the `Credential` parameter passed to all four operations identifies the slice being controlled. The first two operations stop and start the execution of an existing slice. The slice retains any acquired resources on the component, although a component that uses work-conserving schedulers is free to utilize those resources for the duration of the suspension. The slice should not expect the threads running in the slice to resume at the point the slice was suspended, as the implementation of `StopSlice` is free to kill all running threads, in which case, `StartSlice` effectively reboots the slice. However, the slice's on-disk state should remain unaffected by the operations. The third operation resets a slice to its initial state. This includes clearing any on-disk state associated with the slice. Thus, `ResetSlice` is effectively equivalent to deleting and re-creating the slice on the component, but without freeing the slice's resources. The fourth operation removes the slice from the component and releases all of its resources.

**Note: Does a freshly instantiated slice/sliver start in the suspended state (and hence, one must invoke the `StartSlice` operation to "boot" it), or is each active sliver in a slice automatically booted when it is instantiated?**

Note that these operations might be invoked by a user responsible for the slice (e.g., a researcher associated with the slice with the slice or the PI that vouched for the slice), or by a user responsible for the component (e.g., an operator affiliated with the MA). In the latter case, the operator might not know that the slice exists on the component, but is terminating or suspending the slice on all components it manages. This permits an operator to control a slice on all of the components it manages without the cooperation of a slice manager that knows all the components on which the slice has been embedded.

### 6.2.4 Slice Information

Components support two informational operations:

```
SlicesNames[ ] = ListSlices(Credential)
RSpec = GetResources(Credential)
```

The first is used to learn the HRNs for the set of slices instantiated on that component or aggregate; a credential that contains any valid GID is sufficient to make this call. The second is used to either get the resources available on the component or aggregate (if the given credential does **not** correspond to a slice), or get the set of resources bound to a particular slice (if the given credential does correspond to a slice).

In practice, a user calls `GetResources` with his or her user credential to learn what resources are available, next calls `CreateSlice` with a slice credential to ask that resources be allocated to the slice, and finally calls `GetResources` again (this time with a slice credential) to learn precisely what resources the component or aggregate assigned to the slice. This sequence can be repeated to incrementally acquire the desired resources.

Note that when `GetResources` is invoked on an aggregate, the caller is able to learn the set of components available within that aggregate. This information is likely to be both more detailed and more dynamic than the component information available in a registry.

A third operation

`SliceName = GetSliceBySignature(Credential, Signature)`

where

`Signature = (StartTime, EndTime, Protocol, SrcPort, SrcIP, DstPort, DstIP)`

is used to learn the HRN for the slice that sent a particular packet onto the Internet. It is meaningful only on a component that is able to forward packets to/from the legacy Internet.

### **6.3 Component Management Interface**

A component management interface (or simply, “management” interface) is used to boot and configure components, bringing them into a state that they can support the slice interface. The interface is also used to bring the component into a safe state should the component be compromised. Both individual components and aggregates representing a set of components can be expected to support the management interface.

The management interface includes three operations:

`SetBootState(Credential, State)`

`State = GetBootState(Credential)`

`Reboot(Credential)`

The first operation is used to set the boot state of a component to one of the following four values: **debug** (component fails to boot, but should keep trying), **failure** (component is experiencing hardware failure, and so is taken offline until a human intervenes), **safe** (component available only for operator diagnostics), or **production** (component available for hosting slices). The second operation is used to learn a component’s boot state and the third operation forces the component to reboot into the current boot state.

Note that we expect a given component (or aggregate) to support a much richer set of management-related (O&M) operations, effectively extending the required operations listed here. The management interface defines only the minimal set of operations **all** components (including aggregates and proxies) must support.



## 7 Authorization and Access Control

This section outlines the origins and flow of trust throughout an SFA-based system. This includes the expected policies for granting the privileges defined in Section 5.4. In other words, we expect the `GetCredential` operation to return credentials that adhere to this policy.

All rights regarding slices originate with slice authorities. SAs approve of (take responsibility for) slices and the users associated with them. Each SA implicitly has the `authority` privilege for the registry records corresponding to the set of users and slices for which it is responsible. SAs typically grant the `authority` privilege to the PI associated with the authority.

All rights regarding component resources originate with management authorities. MAs define the resource allocation policies for the components they manage and approve of all users that operate those components. Each MA implicitly has the `authority` privilege for the registry records corresponding to the set of users and components for which it is responsible. MAs typically grant the `authority` privilege to the owners and operators associated with the authority.

Users, components, and authorities are granted the `refresh` privilege for the registry record that contains information about them; users also have this privilege for the slices they are affiliated with. All users and authorities are granted the `resolve` privilege for all records in the registry. All users and authorities are granted the `info` privilege for all slices in the system.

Users associated with an SA (i.e., PIs) are granted the `pi` privilege for all slices registered with that SA, as well as for all slices registered by any sub-authority rooted at that authority. This privilege cannot be delegated.

Users associated with a slice (i.e., researchers) are granted the `instantiate`, `bind`, and `control` privileges for that slice. We call these out as three separate privileges so that users can delegate useful subsets of the operations defined by the slice interface to third party services (e.g., the right to control an existing slice). These users will likely disable delegation before passing the credential to such a third party service. All users (researchers) are granted the `info` privilege relative to all slices, and all components hosting slices.

Users associated with an MA (i.e., operators) are granted the `operator` privilege for all components managed by that MA, but not for components managed by sub-authorities rooted at that MA. (Such rights must be explicitly delegated.) They are also granted the `pi` privilege on all components they manage, across all slices hosted on those components. This latter right allows an operator to shut down or suspend any misbehaving slice that its components host.

Each component implements a resource allocation policy that determines how many resources, if any, to grant each slice. A user that is granted the `instantiate` or `bind` privileges for a given slice is viewed as having the right to ask for resources from the component – the credential essentially confirms that some slice authority vouches for the slice – but it is up to the component to decide if it is willing to host the slice, and if so, how many resources to grant it.