Overview of the GENI Architecture

GDD-06-11

GENI: Global Environment for Network Innovations

January 5, 2007

Status: Draft (Version 0.9)

Note to the reader: this document is a work in progress and continues to evolve rapidly. Certain aspects of the GENI architecture are not yet addressed at all, and for those aspects that are addressed here, a number of unresolved issues are identified in the text. Further, due to the active development and editing process, some portions of the document may be logically inconsistent with others.

This document is prepared by the Facility Architecture Working Group.

Editors:

Larry Peterson, *Princeton University*

John Wroclawski, *USC/ISI*

Contributing workgroup members:

Paul Barford, *University of Wisconsin*

Jack Brassil, *HP Labs*

Ted Faber, *USC/ISI*

Alefiya Hassain, *Sparta*

Jay Lepreau, *University of Utah*

Larry Peterson, *Princeton University*

Robert Ricci, *University of Utah*

Steve Schwab, *Sparta*

John Wroclawski, *USC/ISI*

Other contributors:

Tom Anderson, *University of Washington*

Steve Muir, *Princeton University*

Timothy Roscoe, *Intel Research*

Stephen Soltesz, *Princeton University*

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

Revision History

Version	Changes log	Date
v.06	Original version posted	9/12/06
v0.7	Expanded naming discussion (new section 3, 4.1.4, 4.2.1) Expanded component requirements (4.1.3) Expaned rspec discussion (7.1) including two extensive examples (7.1.8) Expanded example deployment (9) Spun glossary into a stand-alone document	11/1/06
v0.8	Reworked naming (section 3) Included example about components registering (8.3) Made terminology consistent with new WBS (throughout)	12/7/06
v0.9	Edited throughout for consistency in security Recast portals so they aren't a kind of aggregate Added network interface definitions (new section 4.1.3)	1/5/07

Table of Contents

1 Introduction

This document discusses the GENI system architecture. It focuses on the set of abstractions and interfaces at the heart of this architecture—the so-called *GENI Management Core* (GMC)—and in the process, gives an overview of the entire architecture.

This document *does not prescribe* a particular deployment of links, nodes, subnets, and so on (although it does sketch an example deployment in Section 9 to help make the discussion more concrete). While we realize that to some readers a "network architecture" implies a specific configuration of devices and a specification of how they are interconnected, such a prescription is not the intent of this document. We expect such details to be defined over time—and continue to evolve—in accordance with scientific requirements and budgetary constraints; the framework defined in this document simply governs how these pieces are logically plugged into a coherent facility.

As the title indicates, this document primarily serves as an overview. A companion set of XSD and WSDL specifications complete the picture by giving precise definitions of the GMC interfaces [GMC Spec].

This document makes use of sidebars to call out related and unresolved issues. They include:

> **Unresolved Issue:** Identifies an unresolved issue or a topic that requires further development.
>
> **Engineering Note:** Addresses an engineering issue, such as performance, reliability, and availability.
>
> **Technology Decision:** Identifies a concrete technology—protocol, format, language—that implements some aspect of the GMC.

Although it addresses issues at a low level, the intent of the GMC is to provide a framework for building an open, large-scale, realistic experimental facility that allows researchers to experimentally answer questions about complex network systems, evaluate alternative architectural structures, and quantify engineering tradeoffs.

2 GENI Architecture and the GENI Management Core

The overall GENI architecture can be divided into three levels, which this note refers to as physical substrate, user services, and GMC.

At the bottom level, GENI provides a set of physical facilities (e.g., routers, processors, links, wireless devices), which we refer to as the *physical substrate*. The design of this substrate is concerned with ensuring that physical resources, layout, and interconnection topology are sufficient to support GENI's research objectives.

At the top level, GENI's *user services* provide a rich array of user-visible support services intended to make the facility accessible and effective in meeting its research goals. As researcher requirements change and GENI's sophistication advances, it is expected that the set of available

user services will change and advance as well. A key goal of the overall GENI architecture is to enable and foster the evolution of these services throughout the lifetime of the facility.

Sitting between the physical substrate and the user services is the GENI Management Core, or GMC. The purpose of the GMC is to define a stable, predictable, long-lived framework—a set of abstractions, interfaces, name spaces, and core services—to bind together the GENI architecture. Because GENI's physical substrate and user services will develop and evolve rapidly as the facility is constructed and used, the GMC is designed to provide a narrowly defined set of mechanisms that both *support and foster* this development and *isolate* developmental change in one part of the system from that in other parts, so that independent progress may be made.

By analogy with today's Internet architecture, the GENI architecture conforms with the hour-glass model: the GMC corresponds to the IP layer of the Internet stack with its attendant addressing routing and service model (i.e., it defines the "narrow waist" of the GENI hourglass); the set of high-level management services corresponds to the additional functionality needed to make the Internet a complete system (e.g., HTTP, WWW, Skype); and the GENI substrate corresponds to the collection of computing and networking devices that make up the physical Internet. This document focuses on the minimal set of primitives that defines the GMC, but in doing so, also outlines the overall GENI architecture.

For GENI to function, the GMC must be reduced to a specific implementation. We refer to this implementation as the GENI Management Core Implementation, or GMCI. The GMCI provides two key elements of the overall GENI system. These are the small set of core services that are necessary for the system to operate, and an underlying messaging and remote operation invocation framework needed for elements of the GENI system to communicate with each other. An important aspect of the GMCI is its ability to allow different implementations and versions of non-core user services to be easily added to the system as appropriate.

> **Technology Decision:** While this note primarily describes the GMC at the architectural level, reference is made where necessary to aspects of the GMCI, including specific data formats, interface specifications, and elements of the messaging framework. The current GMCI is based on selected technologies from the Web Services world, and benefits greatly in function and implementation simplicity from leveraging those technologies.

Note that the GMC is not a management service or operations center, as implied by earlier descriptions of the GENI architecture [GDD-06-07]. The GMC only defines the framework within which such facilities can be constructed.

It is also important to distinguish between the narrowly defined instantiation of GENI that we expect to build as part of an NSF facility construction project (we call this "GENI proper" or "NSF GENI"), and the larger federation of GENI-like facilities (we call this the "GENI ecosystem"). This document defines the GMC in terms of the larger GENI ecosystem (e.g., the GMC explicitly supports federation), although it often uses scenarios motivated by requirements of the NSF-funded project to illustrate specific aspects of the architecture.

2.1 Players

The GMC touches several categories of users and organizations:

- *Owners* of parts of the 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.
- *Administrators* of parts of GENI, often working for owners or under contract to the GENI organization, whose job it is to keep the platform running, provide a service to researchers, and prevent malicious or otherwise damaging activity exploiting the platform.
- *Developers* of infrastructure services who build on the GMC functionality to implement services of general use to the GENI community.
- *Researchers* employing GENI in their work, for running experiments, deploying experimental services, measuring aspects of the platform, and so on.
- *End users* not affiliated with GENI, but who access services provided by research projects that run over GENI.
- *Third parties* that may be confused or concerned about the effect that GENI-hosted experiments and services are having on their own enterprises.

It is not the goal of the GMC to directly address the requirements of these groups of users. In practice, many of these users will interact with the substrate through mediating services. Instead, we can break down the goals of the GMC into various activities aimed at supporting these higher-level services.

2.2 Actions

Based on this list of players, we can identify the following activities that the GMC must mediate:

- Allow owners to declare resource allocation and usage policies for substrate facilities under their control, and to provide mechanisms for enforcing those policies.
- Allow administrators to manage the GENI substrate, which includes installing new physical plant and retiring old or faulty plant, installing and updating system software, and monitoring GENI for performance, functionality, and security. Management is likely to be decentralized: there will be more than one organization administering disjoint collections of GENI sites. A broad spectrum is possible, ranging from individual owners managing their own machines, to a small number of large organizations that federate at a coarse grain.
- Allow researchers to create and populate experiments, allocate resources to them, and run experiment-specific software. Some of this functionality, such as convenient installation of software like libraries or language runtimes, may be provided by higher-level services; in this case the aim of the GMC is to support these services (see next point). The GMC must also expose an execution environment to researcher's applications, experiments, and services. These execution environments must be flexible (i.e., support a wide-range of program behavior) and perform satisfactorily.

- Expose low-level information about the state of the GENI substrate to developers, so they can implement high-level monitoring, measurement, auditing, and resource discovery services. In some sense, GMC can be regarded as analogous to the "kernel" of GENI as a distributed system, and consequently should expose (in a controlled manner) information to services interested in both managing the system, using it effectively, and scientifically observing its operation.

A central goal of the GMC is to provide interfaces through which researchers request resources and owners establish policy regarding limits placed on those resources. These requests/limits include statements about:

- **Resources**: It must be possible to request/limit the amount of processor, storage, memory, and network resources consumed by a user experiment.
- **Privileges**: It must be possible to request/limit the privileged operations that an experiment may invoke. Examples privileges include the right to access measurement sensors [GDD-06-12], and the right of infrastructure services to read and write system state [GDD-06-24].
- **Containment**: It must be possible to request/limit how an experiment interacts with the rest of the network. As examples, this includes what addresses and ports an experiment connected to the current Internet can send to and receive from, the rate of unique addresses to which an experiment can send packets, and the types of packets that can be generated.

3 Naming

The GMC defines unambiguous identifiers—called *GENI Global Identifiers* (GGID)—for the set of objects that make up GENI. These objects, as defined fully in the next section, include users, components, aggregates, and slices.

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

Specifically, a GGID is represented as an X.509 certificate [X509, RFC3280] that binds a Universally Unique Identifier (UUID) to a public key. The object identified by the GGID holds the private key, thereby forming the basis for authentication. Each GGID (X.509 certificate) is signed by the *authority* that created and controls the corresponding object; this authority must be identified by its own GGID. There may be one or many authorities that each implement the GMC., where very GGID is issued by an authority with the power and rights to sign GGIDs. Any entity may verify GGIDs via cryptographic keys that lead back, possibly in a chain, to a well-known root or roots.

> **Engineering Note:** UUIDs are defined by ISO X.667 [X667]. UUIDs may be generated locally using any one of several mechanisms that allow for low collision rates. There is a URI name space [RFC4122] set aside for UUIDs, which simplifies their use with Web Services.

A *name repository* maps strings to GGIDs, as well as to other domain-specific information about the corresponding object (e.g., 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). The GMCI provides a default repository that defines a hierarchical name space for objects, corresponding to the hierarchy of authorities that have been delegated the right to create and name objects. This default repository assumes a top-level naming authority trusted by all GENI entities, resulting in names of the form:

top-level_authority.sub_authority.sub_authority.name

For the purpose of this document, we assume "geni" is the top-level authority, but allow for the possibility that other similar authorities might federate in accordance with the GMC.

Note that the name assignment trust relationship represented by the default GMCI repository is distinct from other trust relationships in GENI. Rights to manipulate GENI objects may stem from different trust relations than the rights to name objects. For example, the right to name components might belong to a systems administrator, while the right to allow access to them might be allocated by the researcher in charge of a laboratory or by the GENI Science Council. There is no requirement that the name assignment hierarchy and these other hierarchies are distinct, but the flexibility of separating them is intended to allow different services to be implemented and to cleanly allocate responsibilities.

> **Engineering Note:** We expect there to be two default repositories: one for components and aggregates of components (called the *component repository*), and one for slices (called the *slice repository*). Users will also have GGIDs, but it's not clear GENI needs to provide a searchable "user repository."

A repository is implemented as a service in GENI, and hence, it is identified with its own GGID.[1] The protocols used to register with or query a repository are described in WSDL and are extensible and exportable. New repositories will be able to describe their capabilities in WSDL and common tools can be used to contact repositories. A detailed example of how a component joins the global component repository is given in Section 8.3.

This document does not constrain the implementation of repositories. They must be able to support retrieval of GGIDs and contact URIs, in response to whatever query format they export via WSDL. Repositories must scale appropriately to their domain, which implies that the default repositories must scale significantly. Their implementations and additional interfaces are unconstrained by this specification. Future repositories need not be hierarchical—e.g., they may be attribute-based—although GGID validation will continue to be constrained by the X.509 certificate structure.

The separation of identity (GGID) from the various present and future system name spaces (repositories) allows for simplicity of the GMC implementation, while encouraging new ways to

---

[1] As will be explained in the next section, a service that runs in a slice of GENI, as is the case with a naming service, can be identified by the name and GGID for the slice.

organize system data. The GMC itself operates on GGIDs—no matter what name a caller uses to retrieve a GGID from a repository, the object the caller contacts will be asked to authenticate itself as the GGID, not the name. New services may create new name spaces without disrupting GMC operations. New interfaces can become visible to the system by using a repository to bind them to a GGID / URI pair.

4 Abstractions

This section introduces the three major abstractions that the GMC defines: *components*, *slices*, and *aggregates*; the following section describes the interfaces they support. This section also talks about the main actors in the system: *users*.

4.1 Components

The GMC defines *components* as the primary building block of GENI. 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 both physical resources (e.g., CPU, memory, disk, bandwidth) and logical resources (e.g., file descriptors, port numbers. 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 (described in Section 6.2). The component manager abstraction defines the operations available to the GMC and higher level services to manage the allocation of component resources to different users and their experiments.

> **Technology Decision:** One concrete representation of the CM interface is that it exports a set of procedures that can be invoked using XML-RPC. While many such access protocols are possible, and we want to define the GMC in such a way that it does not require today's TCP / IP Internet, selecting a specific access protocol is necessary if components are expected to interoperate.

4.1.1 Multiplexing

It must be possible to multiplex (slice) component resources among multiple users.[2] 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 distinct partition of the component's resources). In both cases, we

[2] It is not the case that a component must be multiplex-able across an infinite number of multiple users. Although every component must respond to the slice creation and control operations, each component may establish limits on the number of simultaneous users it can support. In particular, a component that is incapable of being shared may limit itself to supporting one active user at any given time.

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-provide programs can also be viewed as supporting an *execution environment*. Slivers that support such execution environments are said to be *active slivers*.

The precise execution environment provided by a sliver can be defined along several dimensions, and hence, any type system for slivers could become arbitrarily complex. We strongly recommend defining a modest number of "base" sliver types—e.g., virtual server, virtual router, virtual switch, virtual access point—and giving users the ability to customize their environments by installing additional software packages.

A component *owner* is a principal that establishes policies about how the component's resources are assigned to users; that is, owners define the resource allocation policy for their components. Other aspects of being an owner—e.g., establishing acceptable use policies for a given component—are not specified in this document.

> **Unresolved Issue:** Due to the nature of the partnership between NSF and research sites that will host GENI hardware, we may need to distinguish between two activities that are typically the purview of a component owner: (1) deciding how the component's resources are allocated to different purposes, and (2) defining the acceptable use policy for programs that run on the component. NSF (and the Science Council that represents the research community) will need to specify resource allocation policy, while individual sites will need to specify the terms and conditions under which a GENI component may operate within their site.

4.1.2 Containment

It must also be possible to restrict the behavior of a component—and the slivers it hosts—in the surrounding network. This is particularly important with respect to traffic that flows between the component and the legacy Internet. This containment includes the rate at which a component can send traffic into the network, the other components and slivers with which it can communicate, and the rate at which a component can initiate communications with new components. It may also restrict packet-formats and the ability to spoof source addresses, as well as place protocol-specific rate limits on slivers. The component must also audit packets that are transmitted to the Internet so that responsibility for any unwanted or disruptive traffic can be traced back to the responsible sliver.

In the worst-case scenario, it must be possible to rapidly disconnect the component from the network, and bring it into a safe state, from which problems (including potential security compromises) can be diagnosed.

4.1.3 Virtualized Network Interface

Each component offers an interface by which slivers access the underlying network substrate. The GMC initially defines three such interfaces:

- **Virtual Server Interface**: Slivers access the network through the standard socket interface. This interface allows inter-component communication is via TCP or UDP connections, or IP tunnels. The server interface does not imply a fixed topology—all components participating in the experiment are reachable (addressable) to the extent they are reachable via today's Internet.
- **Virtual Router Interface**: Experiments access the network through a virtualized link interface. This interface exposes transmission queues and link failures, and the virtual links can be configured to support CBR guarantees. The router interface implies a fixed topology—the set of links (peers) available at each component is statically defined when the slice is created, and is **not** under control of the experiment running in the slice. There are two sub-cases of this interface: (1) the virtual link is point-to-point between a pair of components; and (2) the virtual link is multi-point, corresponding to a VLAN or a wireless network. We sometimes refer to the wireless version of the second case as a "Virtual MAC Interface."
- **Virtual Switch Interface**: Experiments access the network through a virtualized control interface that permits the dynamic creation, termination, and provisioning of end-to-end circuits. This switch interface does not imply a fixed topology—some amount of physical link capacity (e.g., an optical wavelength) is allocated to a given experiment when the containing slice is created, but the set of end-to-end circuits multiplexed onto that capacity is under the control of the experiment. There are two sub-cases of this interface: (1) the interface includes framing; and (2) the interface does not include framing, implying that the experiment has access to a raw optical wavelength.

Note that one should not read too much into the names assigned to each interface type. They were chosen to reflect the protocol layer in today's Internet that would use the interface. For example, a router that forwards packets between (possibly heterogeneous) links would run on top of the router interface. Said another way, the router interface allows researchers to experiment with a virtual network consisting of a set of virtual routers. The functionality implemented by such virtual routers might be similar to or completely different from today's IP router, the important point being that both need some virtual topology among a set to nodes.

4.1.4 Compliant Implementation

Each component exports a well-defined CM interface (see Section 6.2) through which users create and control slivers on that component. Supporting this interface implies that the component must provide software and hardware mechanisms that implements a defined set of functions, including:

1. create and destroy slivers, bind a set of resources to a sliver, and reclaim those resources;

2. isolate slivers from each other, such that

a) the resources consumed by one sliver do not unduly affect the performance of another sliver,

b) one sliver cannot, without permission, eavesdrop on network traffic to or from another sliver,

c) one sliver cannot access objects (e.g., files, ports, processes) belonging to another sliver, and

d) users are allowed to install software packages in their sliver without consideration for the packages installed in other slivers running on the same component;

3. allow users to securely log into a sliver that has been created on their behalf;

4. deliver signals to slivers, including a "reboot" signal that is delivered whenever the sliver starts up;

5. grant privileged operations to select slivers, including the ability of one sliver to access private state associated with another sliver (thereby supporting sliver interposition);

6. disconnect the component from the network and bring it into a safe state;

7. rate-limit the network traffic generated by a sliver, as well as by all slivers running on the component;

8. for components supporting a connection to the existing Internet, limit (filter) how a sliver interacts with (exchanges packets with) the Internet;

9. for components supporting a connection to the existing Internet, support a mechanism to audit all packet flows transmitted by slivers to the Internet, and determine what sliver (slice) is responsible for a given packet;

Broadly speaking, the first five functions address intra-component requirements (i.e., how slivers interact with the component and each other), while the last four functions correspond to inter-component behavior (i.e., how the component and its slivers interact with the rest of the network). A companion document gives a reference design for a canonical GENI component, illustrating how a component might implement these requirements [GDD-06-13].

Note that the requirement of performance isolation (2a) is purposely underspecified. Ideally, it would be possible to completely and perfectly isolate a sliver from the resources consumed by other slivers—i.e., offer absolute resource guarantees to individual slivers—as well as to provide for slivers that do not require deterministic guarantees. However, two factors make this difficult in practice.

First, the overhead cost of perfect isolation—which at the limit might include providing separate physical resources to the isolated slice—may be very high. In a virtualized system, the response to this problem is to provide isolation to an acceptable, rather than perfect, level of fidelity. This in turn triggers the second limitation, which is that it is not clear how to best define "acceptable," balancing user requirements with the implementation pragmatics and overheads of isolation. There are many scheduler and virtualization options, each of which potentially offer a different level of fidelity to users. Which do we chose?

Rather than attempt to specify a single, global answer to this question, our response is to allow GENI's users to guide us. We provide an architectural *framework*, allowing different levels of isolation to be implemented by different component designs, and allowing users to select components based in part on their isolation capabilities and costs. We believe that over time usage and implementation experience will lead a small but useful set of design points to emerge. To be clear, it is the intent that GENI support both best-effort and reservation-based

usage. What the architecture cannot (and should not) say is with what precision reservation-based guarantees can be made; the answer to that question depends on implementation choices.

> **Engineering Note:** One aspect of performance isolation that requires particular attention is how sliver traffic is multiplexed onto a shared link, thereby supporting networking experiments that need predictable delay and bandwidth. There are two potential implementation strategies. One is to *shape* the traffic emitted by each sliver, and then rely on *FIFO scheduling* of these flows when they reach a multiplexing point. (This multiplexing point might be a device queue within a single component, or a forwarding queue at a switch that multiplexes traffic from multiple components onto a single tail circuit.) The second strategy is to complement shaping with *link scheduling* at the multiplexing point, with the scheduler dividing link capacity among flows / slivers according to any guarantees they were promised. The latter approach implies that control over resource allocation must reach beyond individual components to any shared multiplexing point, but doing so is necessary to ensure (1) efficient use of resources (otherwise best effort slivers will need to behave conservatively), and (2) fine-grained promises can be made to slivers that require them. We believe the second approach is required.

Components must be able to deliver signals to locally instantiated slivers (4), informing them of external events. A sliver may register a handler for each signal. Standards signals will be sent on calls to the slice control API (detailed in section 5.5.2), and on sliver 'boot', which may happen more than once (if the host component reboots, for example.) This gives slivers a hook for initializing their environment, for cleaning up on sliver destruction, and for setting up runtime state, such as daemon processes. Components may also define their own signals, but they must provide default handlers (which may be null). For example, a component may choose to deliver a signal to inform a sliver that it is using more than its soft share of some resource (such as RAM), to give it a chance to voluntarily release those resources.

Note that the requirements related to containment (8 and 9) need only be supported with respect to traffic flowing between a GENI component and the legacy Internet. Thus, the mechanisms that implement these functions can be limited to IP addresses and TCP / UDP port numbers.

Also note that the phrases “install software packages” and “log in” used in items 2 and 3 above are appropriate for components such as general purpose computational nodes, but may not be appropriate for more specialized components.

> **Technology Decision:** While the exact form of requirements 3 and 4 is component-specific, we expect a common approach will be to provide remote access via SSH, and the signal delivery mechanism will execute well-known binary files (e.g., /etc/initrc for the reboot signal).

> **Engineering Note:** While we describe these functions as requirements placed on a component, this is not to imply that they must all be implemented within a single “box,” as an integral part of the component. For example, while functions 6-9 might be implemented by the OS running on a component, they might also

be packaged in a separate device that effectively polices how the component (and the slivers it hosts) interact with the rest of the network. If such a device is positioned at a choke point for an entire network (or administrative domain), it can enforce appropriate externally visible behavior for a set of components.

4.1.5 Component Names

Components are uniquely identified using GGIDs, as described in Section 3. In addition, a default component repository binds a human-readable name to each component according to a hierarchy of *management authorities* (MA) that are each responsible for some set of 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 likely to be responsible for the operational stability of the component, as described in Section 8.2. The repository also stores the URI at which the component's manager can be accessed, along with other attributes (and identifiers) that might commonly be associated with a component (e.g., hardware addresses, IP addresses, DNS names). The top-level management authority, geni, is itself named by a GGID, and a public-key bound to this GGID is used to authenticate communication.

4.2 Slices

A *slice* corresponds to a set of slivers spanning a set of GENI components, plus an associated set of users (researchers) that are allowed to access those slivers for the purpose of running an experiment on GENI. Users run their experiments in a slice of the GENI substrate.

A slice has a name, which is bound to the set of users associated with the slice, as described below. There exists a (possibly empty) set of slivers that participate in the slice, but the GMC does not define a concrete representation for slices. Instead, a slice is defined by whatever service creates the set of slivers on behalf of some user.

4.2.1 Slice Names

Slices are uniquely identified by GGIDs, as described in Section 3. In addition, a default slice repository assigns a human-readable name to each slice, corresponding to the hierarchy of *slice authorities* (SA) that are responsible for the behavior of the slice. For example,

geni.us.princeton.codeen

might name a slice created by the GENI slice authority, which has delegated to the US, and then to Princeton, the right to approve slices for individual projects (experiments), such as CoDeeN. GENI defines a set of expectations for all slices it approves, and directly or indirectly vets the users assigned to those slices. Note that the GENI slice authority is expected to support slice creation on behalf of network and distributed systems researchers as part of an NSF project. Because it is possible that other related facilities will federate with GENI, and there will be other uses of the greater GENI ecosystem, we allow for the possibility that there will be other top-level slice authorities.

The slice repository also binds each name and GGID to a record of information that includes (1) email addresses for the set of users associated with the slice, (2) a set of public keys for each of those users, and (3) other contact information for each of those users. Entities needing this information ask the repository to translate either the slice's name or its GGID into this data record.

Note that slice names are used for many purposes. One, they identify slices to human users. Two, they represent a hierarchy of slice authorities, where we assume a slice name is not both a slice and a slice authority. Three, they represent authorization to create slices and manipulate slices, both in the sense that a user may invoke a slice create operation on a slice name in the slice authority hierarchy (as described in Section 6.1) and in the sense that a user bound to this slice is explicitly authorized to perform all operations on the slice. Four, they identify slices within an audit trail of responsibility. Finally, they index an authorization database of slice authorization keys, and so on.

4.2.2 Experiments

An *experiment* is a researcher-defined use of a slice; we say an experiment runs in a slice. Although often conflated, it is important to distinguish between the two concepts. The GMC defines an interface that is used to create and control slices. This includes operations for embedding the slice in a set of GENI components. Once a slice exists and resources have been bound to it, a researcher is free to further configure (program) the slice as part of an experiment, either directly, or by utilizing available experiment management services. This includes loading and executing code in the slice's constituent slivers, and setting any parameters exposed by the constituent execution environments. Thus, slice creation is distinct from experiment configuration, and in fact, multiple experiments may run in a single slice over time, each parameterized in a different way but all utilizing the same set of component resources.

Many experiments are expected to be short-lived, for example, they might run for an hour at a time. Some experiments will be longer-lived, running continuously and processing real network traffic on behalf of a user community. Because such experiments are likely to provide a useful service to some user community, we often call them *experimental services*. In many respects, these experimental services are similar to the *infrastructure services* described in the next subsection, the only difference being the former are likely targeted at end users not affiliated with GENI, while the latter are specifically designed on behalf of GENI researchers.

4.2.3 Services

We expect the set of distributed services that users employ to help manage their slices, and leverage to help implement the experiments running in those slices—collectively called *infrastructure services*—will themselves often run in slices of GENI. For example,

geni.monitor

geni.filesystem

geni.repository

might denote GENI-wide monitoring, file system, and repository services, respectively. All three of these services run in their own slice of GENI and export an interface to other GENI users. Note that similar services can be provided by any SA in the system; e.g.,

geni.us.utah.monitor

might correspond to an alternative monitoring service. Because it is often the case that slice names and service names are equivalent from a user's point of view, we sometimes use the terms “slice” and “service” interchangeably in this document.

> **Engineering Note:** The notion that all services are embedded in slices creates the possibility of a booting dependency conundrum. It must be possible to boot from scratch in case of complete failure or security shutdown (a claim that cannot necessarily be made about the current Internet).

When a slice name is used to also denote a service, the slice name repository likely includes a URI at which an interface for the service can be called. Note that there are likely to exist services that do not run in a slice, for example, by supporting the composition of other services. We sometimes call an interface constructed from the composition of sub-services a *portal*. For convenience, we allow for the possibility that these services (and portals) might also be named by a slice / service name.

4.3 Aggregates

While the architecture does not mandate any particular relationship among components—users are free to create a set of slivers on any set of components for which they can acquire the necessary rights—for a variety of reasons it is useful to define functionality relative to a related set of components. We introduce an *aggregation* construct for this purpose.

An *aggregate* is a GENI object that represents an unordered collection of components. It has an identity and supports the natural operations of a collection (e.g., addition, deletion, and enumeration). Aggregates provide a way for users, developers, or administrators to view a collection of GENI components together with some software-defined behavior as a single identifiable unit.

Aggregates can be hierarchical, that is, they can contain other aggregates as well as components. A given aggregate or component can be a member of zero, one, or many aggregates.

Example aggregates might correspond to a physical location (components co-located at the same site), a cluster (components that share a physical interconnect), an authority (a group of components managed by a single authority), a configuration (a group of components that share configuration information), or a network (a group of components that collectively implement a backbone network or a wireless subnet). There also might be a “root” aggregate that corresponds to all GENI components.

An aggregate is an active object, able to accept and respond to operations invoked on it. These operations are implemented by an *aggregate manager* (AM). Similarly to a component and its component manager, an aggregate has a distinguished GGID, used by the GMCI to invoke

operations on the aggregate, and it has a human-readable name drawn from the same name space as components.

> **Unresolved Issue:** To the extent that an aggregate exports a component interface, it can be treated as a component, and there's a good argument for naming it as such. However, it seems clear that aggregates will also export other interfaces that components will not—e.g., the aggregation interface—and a reasonable model for that needs to be agreed upon. Two possibilities that probably lie at ends of a continuum are to name each interface—a component interface sits in the component name space, an aggregation interface is in another—or to name aggregates in one space and provide a reflection interface to ask about other interfaces that the aggregate exports. If we name interfaces, there may be difficulties reasoning about different interfaces to the same collection of underlying components. Naming the aggregates themselves requires deciding how to characterize the interfaces.

A more detailed discussion of how the component and aggregate abstractions can be used in practice can be found in a companion document [GDD-07-42].

4.4 Users

GENI users create and manipulate slices. Each user is identified by a certificate and key pair issued by one of the GENI authorities, where the ability to credential users is delegated. Rights to operate on slices are associated with user credentials. User credentials are associated with slices as described above.

> **Unresolved Issue:** We need to address how component managers reconcile the set of users bound to a slice. The working assumption is that all users affiliated with a slice are granted the same privileges on all components. A related issue is how a component learns what users are affiliated with a slice, which becomes more interesting when the set of users bound to a slice changes.

5 Authorization and Security

The principal role of the GMC is to provide low-level access and control of GENI system resources, for the benefit of users and higher-level services. A key aspect of this role is ensuring that appropriate authorization and security structures exist to protect these services. This section gives a brief overview of the authorization model supported by the GMC.

Two primary classes of resources are protected by GMC-based authorization mechanisms. The first of these is slices, the containers for user-level experiments. The second is the collection of physical and logical resources implemented as, or made available by, components.

5.1 Authorization Model for Slice Creation

Slices are represented in the GMC by names; the creation of a slice name—bound to a GGID—creates the slice. Thus, the authorization model for slice creation is based on controlling access to the slice name space defined in Section 4.2.1. As described there, the slice name space consists

of a naming hierarchy rooted at a top-level Slice Authority (SA). Below the top level SA, intermediate nodes in the naming tree represent intermediate slice authorities, while the leaf nodes represent the slices themselves.

To support authorization, each SA is expected to implement an access control policy on its name creation function. Thus, before creating a sub-node of its portion of the name space, a slice authority must check the credentials of the calling user, and any other information that it sees fit. It is assumed that each SA has the ability to make authorization decisions for its portion of the slice name space directly, without having to refer creation requests to a higher level authority.

> **Unresolved Issue:** This description raises the question of using the scope of slice names in resource allocation authorization. At present, there's no explicit discussion of the possibility of using the scope of a name as part of the resource access control, for example, that components owned by UCLA might choose to grant (some) resources only to slices in the geni.us.ucla name space. In fact, there's no notion that component owners and slices share the same name space hierarchy, so this sort of implicit authorization restriction is not necessarily possible.
>
> An alternative and more general model is that a component, presented with a resource allocation request (that includes a slice name) might, in handling the request, obtain credentials from the SAs that comprise the slice's name hierarchy, as well as from the users that are bound to the slice name. Thus, a component owner might implement an authorization policy basing her allocation decisions on the joint rights of a SA and the users making the request. A range of authorization policies and mechanisms can be supported, with the central feature being the reliance on using appropriate cryptographic checks to authenticate the GENI entity standing behind the request, as well as the authorization or delegation of the rights to make use of the specified GENI resources.

5.2 Authorization Model for Physical Resources

GENI physical resources are encapsulated as components, as described in Section 4.1. Each component has an associated component owner. The owner is responsible for defining and implementing the authorization policy governing use of the component.

The GMC defines a GENI-specific authorization mechanism called a *ticket* used to implement this model. A ticket represents a principal's right to (1) create a sliver (perhaps with some initial resources bound to it) on a component, and (2) bind component resources to an existing slice. Both rights can be delegated. We call the representation of a set of resources a *resource specification* (RSpec). Hence, a ticket is essentially an RSpec signed by a component owner, granting rights to allocate resources on one or more components. RSpecs and Tickets are discussed further in Section 7.

6 Interfaces

This section defines the set of interfaces by which principals manipulate components, slices, and aggregates. The GMC views components and aggregates as objects on which operations are invoked. All operations are invoked relative to a specific component or aggregate; the relevant object is assumed in all specifications given below.

6.1 Slice Operations

In the current GMC architecture slices are instantiated and represented by slice names, that is, the act of creating a slice name is indistinguishable from the act of creating a slice. Thus, slice operations are essentially operations performed by the GMC's slice naming service.

The GMC slice naming repository supports three operations:

GGID = CreateSliceName(Name, UserInfo)

FreeSliceName(Name)

UserInfo = ResolveSliceName(Name)

The first operation creates a Name for a slice and binds it to the associated UserInfo, and the second operation deletes the given Name. The third operation returns the UserInfo bound to the named slice.

> **Unresolved Issue:** The preceding description is overly simplistic. The operations must include the appropriate authorization, preserving the trust relationship among slice authorities, as outlined in Section 5.1 This needs to be fleshed out.

Once a slice name has been registered, the slice exists "in name only." The slice must also be instantiated as a set of slivers before it can be used by a researcher to conduct an experiment on GENI. This instantiation is performed by invoking operations on components on behalf of the slice, as described in Section 6.2

6.2 Component Operations

Once a slice name has been registered with a trusted slice authority, the user can instantiate and provision the slice with the following operations. Although a particular user invokes these operations, it is reasonable to think of the slice as an "actor" that acquires, redistributes, and uses component resources; the users affiliated with the slice are effectively making requests under the name of the slice. We sometimes take the liberty of letting slices be the actors (representing user principals) in the following description.

> **Unresolved Issue:** Note that this decision carries significant implications for the authorization model and implementation. Because a slice is in effect the principal behind requests, users must have a way of delegating a subset of their privileges to a slice. The alternatives are inflexible: binding all privileges to a slice at creation time, granting the slice the union of all privileges of all users' bound to it, and so on.

Note that a single component is able to create only a single sliver, meaning that the following operations must be invoked on each component that the slice is expected to span. We return to the issue of how a high-level service might assist the user in creating a slice that spans multiple components in Section 8.1.

6.2.1 Obtaining Rights to Component Resources

A slice-as-principal invokes the following operations on a component:

Ticket = GetTicket(Slice, Request)

to acquire rights to component resources. The Slice argument is a registered name of a slice. The Request argument indicates (in the form of an RSpec, described in Section 7.1 what resources the slice wants. The returned Ticket effectively binds the slice to the right to allocate on the component the resources / rights indicated by the Request. Whether or not the call succeeds depends on any access control implemented by the component, the local resources available on the component, and the allocation policy implemented by the component.

Two timestamps are associated with each ticket. One indicates how long the ticket is valid. The ticket must be "redeemed" (using the CreateSlice operation described below) within that lifetime. The second indicates a period of time during which the resources specified in the ticket are available to be consumed by the slice.

> **Unresolved Issue:** We also want to support a GrantTicket operation, whereby a component proactively delegates the right to a set of resources to some principal, such as a user or a slice.

Once a principal possesses a Ticket, it can create a sliver on the component by invoking:

Sliver = CreateSlice(Ticket)

The sliver effectively becomes part of the slice identified by the Ticket. In this case, the Ticket must include the "create" privilege. The returned Sliver is a certificate that grants the holder the right to invoke other operations (see below) on the sliver. Note that while this call creates a sliver, we call the operation CreateSlice because it effectively instantiates the slice on that component.

> **Unresolved Issue:** Note that GetTicket+CreateSlice could be combined into a single InstantiateSlice operation.

> **Engineering Note:** To implement the CreateSlice operation, the CM likely invokes local OS system calls. However, network device that cannot be easily modified to support the CM interface might have to be teamed with a general-purpose processor that exports the CM interface, and in turn configures the device through some proprietary interface.

Resources can be bound to a slice—and the duration of their binding extended—via the

ModifySlice(Ticket)

operation with a new Ticket. The consequence of such a call is to augment the slice identified in the ticket with the additional resources specified in the ticket's RSpec; the Ticket must include the "bind" privilege. Similarly, resources can be taken away from a slice. In both cases, a signal indicating how the resource allocation has changed is delivered to the sliver.

> **Unresolved Issue:** Alternatively, components could support distinct Acquire and Release operations.

A component (or other principal) that holds a ticket may need to split off a portion of the corresponding resources to respond to a GetTicket call. The holder of the ticket may have to go back to the original source to do this. This is really just another way to acquire a new ticket, hence

```
Ticket' = SplitTicket(Ticket, Request)
```

where Request says how much of the original Ticket to split off into the new Ticket'; the original Ticket is "reduced" by the same amount. Other rights are preserved.

6.2.2 Controlling a Slice

Component managers also support the following operations on slivers:

```
StopSlice(Sliver)
StartSlice(Sliver)
DestroySlice(Sliver)
```

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. These two operations are often used in tandem to "reboot" a slice, but are offered as a pair to give any management service assisting the slice an opportunity to "clean up state" before restarting. The final operation removes the slice from the component and releases all of its resources.

The Sliver argument for all three operations corresponds to the value returned by CreateSlice. However, any principal with a certificate granting it the right to invoke the operation on a given slice may do so; these operations are not limited to the user that created the slice.

6.2.3 Slice Information

Finally, any principal may invoke the following operations:

```
Names[ ] = ListSlices( )
StatusSpec = GetStatus( )
Name = GetResponsibleSlice(PacketSignature)
```

on a component to learn the names of the set of slices instantiated on that component, the status of the component, and the name of a slice responsible for sending a packet with a given signature, respectively. A PacketSignature is given by a (time, protocol, source IP, source-port,

destination-IP, destination-port) tuple. Note that the GetResponsibleSlice call can only be invoked relative to a packet to/from a GENI component and the legacy Internet.

> **Engineering Note:** The above description assumes that protocol is UDP, TCP, or another protocol using source and destination ports. The format of a PacketSignature may need to be generalized.

6.3 Aggregate Operations

All aggregates support the following operations:

```
Components[ ] = ListComponents( )
AddComponent(Component)
DeleteComponent(Component)
```

Aggregates typically extend this interface with a set of aggregate-specific operations.

Note that the above operation signatures imply that the elements of an aggregate are always of type Component, thus implicitly resolving the ambiguity about whether an Aggregate is in fact also a Component—it is.

7 System Data Objects

This section discusses three key system data objects: RSpecs, Certificates, and Tickets. RSpecs ("Resource Specifications") are used to describe GENI physical substrate resources; the basic elements of the GENI system. Certificates are used to identify principals and allow various data objects and messages to be signed. Tickets, as discussed previously, are used to represent the allocation of rights to a particular resource or system element to a particular slice for a period of time.

7.1 Resource Specifications

RSpec is the data structure used to describe resources in GENI. As such, it contains data about the resources possessed by components, such as their processing capabilities (such as processor architecture and speed), their network interfaces (including bandwidth and the like), and the instrumentation available on them (such as packet logging capabilities). The purpose of the RSpec standard is to give component managers, resource discoverers, slice embedding services, and user front-ends a common resource vocabulary.

An RSpec describes a component in terms of the resources it possesses. Each resource (such as a processor or a network interface) is identified by an identifier (RID), which is assigned by the component manager and must be unique within the component. Thus, a combination of a component name and an RID is sufficient to unambiguously identify any resource in GENI.

RSpec is also used to describe the relationships between components, most significantly, the network links connecting components. This will require the ability to reference resources in other components' RSpecs, which, as noted above, can be done unambiguously. At this time, the

component description role of RSpec is firmer than the relationship specification role of RSpec. Thus, the latter is still subject to change.

> **Unresolved Issue:** The exact nature of network link relationship specification is still undecided. One possibility is to make links themselves first-class components, associating attributes like bandwidth and delay to the link itself. This allows us great flexibility in describing links. However, it is unclear if it makes sense to have a Component Manager for each link. An alternative is to describe the characteristics of the components' network interfaces, then represent links with a simple "connects-to" relationship. This is a simple and intuitive link description, but the downside is that it might make certain properties that are properly associated with the link, rather than interfaces (such as latency) difficult to express properly.

Most users are not expected to use RSpec directly. In this sense, it can be thought of as a "machine language," used below the user-visible level of GENI. It should concentrate on clarity of definition and expressiveness, with lower priority given to user-friendliness. As with any assembly or machine language, some users may choose to use this language directly, but we expect that most will use some high-level interface, such as a GUI, SWORD, or Emulab's extensions to the ns-2 language. Such high-level specifications will be "compiled" to an RSpec by the service or aggregate that offers it. The GMC architecture will support a wide variety of such higher-level embedding services. The ability for higher-level services to provide a user-friendly interface relies on RSpec syntax and semantics having the desirable properties of an output language for machine-compiled higher level specifications, so that it will be a suitable target for a range of front-ends.

Much of the complexity of the GMC is embedded in resource specifications. RSpecs are the "extensible" part of the GMC interface. This document describes a "base" RSpec definition that covers a wide range of resources. However, as new resources and capabilities are added, these specifications will inevitably need to be extended. We expect these extensions will leverage a hierarchical name space: new communities that federate with GENI will be able to extend RSpecs within their own partition of the name space, and components that offer specialized resources will similarly extend the RSpec in their own name space.

> **Technology Decision:** Because many there will be many CM implementations, it is essential that both the syntax and semantic interpretation of the RSpec be clearly defined and easily understood. Further, because the GMC architecture must facilitate the creation of CM implementations across a wide variety of programming environments, the creation of RSpec parsers, validators, interpreters, and similar processors must be straightforward and ideally automatic.
>
> The RSpec definition uses the W3C XML Schema Definition Language (XSD) to describe the structure and typing of RSpec, meeting each of these goals simultaneously. XSD allows for specification of both the syntax and some semantic aspects of the RSpec, and allows for the automatic generation of parsers and tools in a wide variety of programming languages and environments.

7.1.1 Use Scenarios

RSpecs can be used for three distinct purposes. These are:

- *advertising* a set of potential resources. A component manager will use an RSpec to tell potential users of the component what its capabilities are.
- *requesting* a set of desired resources. Users will use RSpecs to make resource requests of components.
- *promising* a set of available resources. Component managers will utilize RSpecs to inform users of commitments made by the component manager to make resources available to the user.

The last case is distinguished from the first by including an "allocate" right within the RSpec. Moreover, because it involves a privileged operation, it is must be signed by the issuer (i.e., the RSpec is encapsulated in a ticket). Such an RSpec is valid only for the component that issued it, and by issuing it, the component promises to honor the RSpec for its specified lifetime. The CM must keep track of all such promises that have not yet expired, and each one should be assigned a unique identifier. It is important to note that honoring an RSpec is not the same as guaranteeing service. An RSpec might represent best-effort or fair-share resources in addition to guaranteed resources. See Section 8.1 for a discussion of resource allocation in GENI.

> **Unresolved Issue:** We hope to use the same RSpec definition for all three uses of RSpec; however, it is possible that we will find that the requirements for the three uses are sufficiently different that the three types will need different definitions.
>
> In particular, the "advertising" role seems likely to require additional information to be presented; attributes of the component that describe its behavior and capabilities but are not "resources" that can be requested or allocated. An example of this would be a description of the execution environment available on a component, such as the instruction set used by the processor. In this case, the choice is between extending the RSpec to describe attributes that go beyond resources, or to incorporate the RSpec as one element of a larger data structure. The first option eliminates the need for a separate "ASpec" data structure, but may significantly complicate the definition of the RSpec. The second alternative allows keeping the basic "vocabulary" of the RSpec the same across all uses, but results in the creation of an additional ASpec datatype to carry additional information in the advertising scenario.
>
> **Unresolved Issue:** It is not clear that the "allocate" right discussed above is needed, or whether it should be part of the RSpec even if so. An RSpec contained within a Ticket signed by a component is distinguishable as a promise, with no additional rights indication required. Further, the "promise" attribute appears to be more closely associated with the ticket itself than with the RSpec.

7.1.2 Scope and Expressiveness

The desired range of expressiveness of an RSpec design must be decided in three separate dimensions. These are level of abstraction, scope of the described parameters, and granularity. This section defines each dimension and discusses it briefly.

Level of Abstraction

As discussed above, the RSpec is intended as a "machine language" for the resource allocation process. Thus, RSpecs are designed to express resource availability and needs at a detailed, concrete level.

We must be careful about the level of abstraction we build into Rspec. It is perhaps tempting to extend the RSpec to express resource requirements at high levels of abstraction, to create a unified format for all purposes. However, languages suitable for expressing resource embedding goals at a high level may be quite different than those of a machine language, for example, procedural rather than declarative, or based on measured performance rather than concrete specification of specific resources.

On the other hand, RSpecs must contain some level of abstraction to avoid embedding assumptions about current networking technologies. For example, RSpec does not limit itself to the current set of link-layer technologies (e.g, Ethernet, ATM), and in fact avoids requiring the use of the OSI network stack model.

This complexity and richness of possible high level resource embedding languages, together with the rapid level of progress in this area, leads to strong conflict with the "universal machine language" requirements of definitional clarity and simplicity of implementation. Thus, it is desirable to avoid the temptation of making a single specification language suitable for all possibilities. The GMC architecture instead allocates to the RSpec the role of universal machine language, providing a framework and support for a number of higher-level embedding services and their respective input formats.

Scope of Described Parameters

A wide range of parameters, ranging from low level resource requirements to detailed configuration of the experimenter's software environment, could potentially be included in the RSpec definition. However, separation of concerns suggests that the GMC architecture distinguishes between the tasks of resource acquisition and environment configuration, because of the wide variety of possible software environments and experiments that might potentially be installed on a similar set of resources.

For this reason we separate the process of acquiring a suitable set of resources (resulting in the creation of a slice) from the process of configuring a specific experiment to run on those resources (i.e., to run in that slice). The GMC architecture views the first as within its scope, while viewing the second as to be handled at a higher level. Thus, RSpecs support the former, but not necessarily the latter. We refer to services that handle the latter as *configuration managers.*

Granularity

The granularity of an RSpec might range from a single component to describing the resource requirements of a complete slice. The base case, discussed here and presently defined in [GMC Spec] specifies at a low-level a single sliver to be instantiated on a specific component.

At the other end of the spectrum, it appears desirable to have a larger granularity, but still low-level, resource allocation language that will allow a user or configuration service to specify and set up the resources of a slice across a possibly heterogeneous set of components, possibly even across all of GENI. While such a description may be viewed as a ``slice-RSpec", the best approach to encoding and representing such a description (e.g., procedural vs declarative) is not yet defined by this specification.

7.1.3 Translation of RSpecs

One RSpec may be compiled into another (or into a set of RSpecs); such usage is expected to be common. Allowing RSpecs to refer to components at multiple levels of abstraction enables a variety of slice embedding and resource allocation schemes. An RSpec may be passed through a set of aggregates (e.g., resource allocators), each of which binds a portion of it to specific resources, allowing each specialized allocator to deal only with the resources it targets.

> **Unresolved Issue:** Wording and viewpoint of the above paragraph appears to be somewhat in conflict with the "machine language" model of RSpecs, even if one fully accepts the recursive aggregator model. It may be the case that the discussion below in the Slice Embedding Services subsection deals adequately with this conflict, and should be elevated here.

7.1.4 Sliver Types

A key attribute of an RSpec is the *type* of sliver that is to be instantiated on the component; the set of applicable resources described by the RSpec will be dependent on this type. The initial base types correspond to the virtual network interfaces defined in Section 4.1.3: (1) virtual server, (2) virtual router, and (3) virtual switch.

RSpecs may be extended in two distinct ways. The primary method of extension is through *subtyping.* GENI component providers will have the ability to extend these basic system types by defining *subtypes.* In this way, components can provide extended capabilities, but those who do not support or do not wish to use the extended capabilities will still understand the basic capabilities and be able to use the component. This ability to extend components in a backward-compatible fashion is the key advantage of subtype-style extension.

A second method of extension is to define new *base sliver types.* In this case, a new type of system element, rather than an extension of an existing type, is being defined. Creating such a definition implies describing the set of resources appropriate to the new type, and making this definition available to the system.

> **Unresolved Issue:** There probably needs to be some discussion of definition scope here; as well as possible architectural mechanisms to formally define base

types (including the "standard" ones mentioned above) and extensions, and to make these definitions known.

7.1.5 Resource Types

RSpec divides resources into four abstract categories: *computation*, *communication*, *storage*, and *measurement*. Most components will have some combination of these resources, and many will have all four. Each resource type is associated with a set of metrics; we have begun by defining a base set of metrics which should be understood by all CMs and services that wishes to interact with a CM. The base set of metrics used for computation, communication, and storage are well known and widely used: for example, storage is defined by its size in bytes, its persistence, whether it is writable, and the (average) latency to access it. Ranges are also supported; one can describe, for example, the range of spectrum available to a wireless device.

Base metrics for monitoring are harder to define, since monitoring is less entrenched in current network architectures than the three other types of resources. GENI, however, will raise them to become first-class. We have not yet defined a set of base metrics for monitoring; we will do so as the GENI monitoring architecture progresses [GDD-06-12].

It is important to keep in mind that these base metrics are *not* the only ways to describe a resource. They are a building block upon which more complicated resource descriptions can be built.

7.1.6 Constraints

As well as specifying values for specific resources, RSpecs can specify *constraints* among possible values for different resources. This is important because many components can support a range of possible resources, but not all at the same time. Similarly, some components will have the property that a certain resource can be allocated only if another resource is also allocated at the same time.

Some example uses of constraints are given below. The reader should refer to the formal RSpec definition [GMC Spec] for the full definition of the RSpec constraint language.

RSpec includes, at its base level, support for declaring sets of mutually exclusive groups of resources. Consider, for example, a wireless network interface that can participate in an 802.11b network or an 802.11a network, but not both at the same time. The component can advertise these two configurations, wrapped in an <or> clause. (Note that we are using the exclusive sense of OR).

In addition to allowing specification of mutually exclusive resource capabilities, this mechanism also allows an RSpec to represent its resources at multiple levels of abstraction. For example, a component may advertise itself as having a general-purpose CPU with some amount of attached storage and communication interfaces. Alternatively, it could advertise itself as a packet-classifying engine with performance measured in packets or bytes per second. These two views can be wrapped in an <or> clause.

Each branch of an <or> clause is identified with its own RID, which must be unique within the scope of the RSpec. This allows for references to the RSpec to unambiguously identify which of the advertised alternatives is being referenced.

Resources can also be grouped in <and> clauses, indicating that all of the advertised capabilities can be used at once. For example, a component can advertise that it has several computational resources (i.e., multiple CPUs). Each resource is given a unique identifier, so that specific ones (i.e., specific network interfaces, CPUs or disks) can be unambiguously identified.

7.1.7 RSpec and Slice Embedding Services

A Slice Embedding Service (SES) is a service which, given some experimenter-supplied network description, selects an appropriate set of components for the experiment. This goal of this section is not to provide an exhaustive description of an SES, but to show how SESes will interact with RSpec.

The process of selecting appropriate resources can be thought of as an annotation. The user presents an SES with a request RSpec (which will commonly describe many desired components) with resources unbound to any particular physical resources. The embedding service selects appropriate resources, annotates the request RSpec with resource bindings, and returns it to the user. The SES need not necessarily bind all resources in the request RSpec; in this way, we can easily support multi-pass selection techniques and specialized SESes.

There is value in allowing SESes to specialize for individual resource types, and in fact, this is the direction in which research on embedding algorithms has proceeded. Emulab's `assign`, for example, specializes in network environments in which all topological components (bridges, routers, trunks, etc.) are known to and allocatable by the embedder. In contrast, SWORD caters to environments in which the entire network topology is not available, such as the Internet. Instead, it uses pairwise measurements of characteristics such as bandwidth and latency to select desirable components. It seems certain that embedding services will also arise for wireless networks, taking that domain's specific challenges into account. A slice using many types of components (ie. wireless, backbone, and edge node) can pass its request RSpec through a series of embedding services, with each binding some different subset of requested components to physical ones.

Some SESes may be capable of not only annotating a request RSpec, but augment them as well. As an example, consider the case of a slice which uses two wireless networks at different sites, and wishes to connect them through the GENI backbone. The experimenter likely does not wish to specify completely the set of routers, links, etc. which the inter-site connection must traverse; instead, the experimenter would prefer to simply specify some high-level properties of the link (such as its bandwidth and a bound on its latency), and leave the selection of specific backbone components to some service. This differs from the previously-discussed slice embedding problem in that there is not necessarily a one-to-one relationship between resources in the request RSpec and physical resources required to instantiate the slice. For example, a single requested inter-site link may, in fact, have to traverse two tail circuits and several backbone routers. An SES that specializes in creating tunnels through the GENI backbone may perform the task of translating the single requested link into the series of physical links it must traverse.

Unresolved Issue: Instead of augmenting the request RSpec, the tunnel service could instead create the tunnel and bind the requested link to that tunnel, maintaining a one-to-one relationship between requested components and the components they are bound to. In essence, the tunnel service (which may be running in a slice) exports both an SES interface and the Component Manager API. This may be a way to avoid translating or augmenting RSpecs, but has the disadvantage that it confounds resource discovery with sliver creation. It may be the case that both methods should be supported. This also raises the larger issue of whether or not slices can or should be able to act as Component Managers. There does not seem to be any reason that they should be prevented from doing so, and in fact, this seems to provide a very powerful virtualization technique.

7.1.8 RSpec in Context

This section puts the RSpec in context by giving examples for representative components. The examples utilize XML syntax: the schema for RSpec types and commentary are available at [GMC Spec]. It should be possible, however, to understand the gist of these examples without reading the schema.

Example 1: Wireless Access Networks with a Backbone Link

For our first example, we consider an experiment in which a researcher wishes to create two wireless access networks at different sites, and connect them through the GENI backbone. Site A, at the University of Utah, contains a number of access points, and will be used by numerous real mobile users, i.e., users with laptops, PDAs, and/or cell phones. Site B, at Rutgers, contains both access points an a set of GENI components that will be used to generate packets on the access network at site B. The experimenter wishes to connect these two sites with a "virtual T3" across the GENI backbone, that is, a link with a guaranteed bandwidth of 45Gbps.

For Site A, the RSpec might look like this:

```
<component type="virtual access point" requestID="siteA-ap1"
        physicalID="geni.us.utah.wireless.node45">
    <!-- Ask for 1GHz of processor power on a general-purpose
         processor -->
    <processing requestID="cpu1">
        <power units="CyclesPerSecond">
            <value>1000000000</value>
        </power>
        <function>Full</function>
    </processing>
    <!-- Ask for 10 GB of storage on the nodes' disk -->
    <storage requestID="disk1">
        <capacity units="GB">
            <value>10</value>
        </capacity>
        <access>R/W<access>
    </storage>
    <!-- Here, we use a hypothetical 'wireless' name space, which extends
         the standard 'communication' type with wireless-specific
         information. The tags without a name space come from the base RSpec
         definition. Here, we ask for an 802.11g network on channel 16. -->
```

```
    <wireless:communication requestID="nic1">
        <medium>FreqShared</medium>
        <mediumtype>broadcast</mediumtype>
        <wireless:protocol>802.11g</wireless:protocol>
        <wireless:frequency type="802.11channel">16</wireless:frequency>
    </wireless:communication>
    <!-- There is a fourth major resource type, measurement, but its syntax
         is not yet defined -->
</component>
<component type="virtual access point" requestID="siteA-ap2"
        physicalID="geni.us.utah.wireless.node42">
... Repeat for each desired access point ...
</component>
```

There are a few things to notice about this example. First, each resource is identified with a unique requestID, so that it can be unambiguously referenced. Setting the physicalID for a resource (in this case, the entire component) indicates that a specific resource is being requested; omitting it allows a slice embedding service or component manger to chose a resource on the user's behalf. Naming for components is still under discussion, and thus the name that appears above is merely a placeholder which may change in later versions of this document. Third, this RSPec uses a hypothetical "wireless" name space to extend the base RSpec with attributes specific to wireless nodes. This name space (not yet defined) would be under the control of the Wireless Working Group or some entity that it delegates. Finally, note that mobile users do not appear in the RSpec: RSpecs describe allocatable resources, which users are not. If, however, slivers were to be run on the user devices, then they could appear in an RSpec.

The RSpec for Site B may look like this:

```
<component type="virtual access point" requestID="siteB-ap1">
    ... Contents omitted for brevity; Similar to SiteA-ap1 above
</component>
<component type="virtual node" requestID="siteB-client1">
    <!-- This component will be used as a client to generate load on
         the access network at site B. -->
    <processing requestID="client1-cpu">
        <!--  We don't require a general purpose processor, merely one
              that can generate packets -->
        <function>Generate</function>
    </processing>
    <storage requestID="client1-disk">
        <!-- Our packet generator will use some small scratch space,
             but this temporary state does not need to be
             persistent -->
        <capacity units="MB">
            <value>5</value>
        </capacity>
        <access>R/W<access>
        <persistence>transient</persistence>
    </storage>
    <wireless:communication requestID="client1-nic1">
        <medium>FreqShared</medium>
        <mediumtype>broadcast</mediumtype>
        <wireless:protocol>802.11g</wireless:protocol>
    </wireless:communication>
 </component>
```

Note here that we have omitted the physicalID so that site B's component manager can chose physical resources for us, and have not requested a particular 802.11 channel, again allowing the CM to chose any available one.

Finally, here is what the RSpec for the cross-site link might look like. As noted earlier, RSpecs for links are very much a work in progress, so the final version may look substantially different. The purpose of this example is to illustrate the types of things that may be expressed about a link.

```
<link type="point-to-point" requestID="siteA-siteB-backbone">
    <endpoint requestID="siteA-endpoint"
        physicalID="geni.us.utah.gateway">
    </endpoint>
    <endpoint requestID="siteB-endpoint"
        physicalID="geni.us.rutgers.gateway">
    </endpoint>
    <medium>SpaceShared</medium>
    <mediumtype>PointToPoint</mediumtype>
    <!-- Our virtual link should have exactly 45 Mbps of (guaranteed)
         bandwidth -->
    <capacity units="Mbps">
        <value>45</value>
    </capacity>
    <!-- We want a maximum of 25 ms latency (one-way) -->
    <latency units="ms">
        <range>
            <min>0</min>
            <max>25</max>
        </range>
    </latency>
</link>
```

Note that we have requested that this link be between a set of gateways, one at each site. However, it is unlikely that a direct link between the two sites exists, and thus a "cut through" will have to be established on the GENI backbone to provide this experiment with the illusion of a virtual T3. We will submit this part of the RSpec to a slice embedding service or a tunnel service, which will find a suitable set of substrate resources to instantiate this link.

Example 2: User-programmable Routers Connected by the Backbone

In this example, we ask for two backbone routers. We want specific routers, connected by a lambda. Each router has a number of programmable network processors; we will be asking for one on each router. The routers also have some memory space available for queuing outgoing packets, and we will request a certain amount of that queuing space. Like the example above, this example assumes the existence of a domain-specific 'routing' extension to RSpec, most likely under the control of the Backbone Working Group. Note that this example does not yet specify how these two routers are to connect to other GENI components or the legacy Internet.

```
<component type="virtual router" requestID="router1"
        physicalID="geni.backbone.pop.seattle.prouter1">
    <!-- We want a network processor on this router, but don't care
         specifically which one we get -->
```

```
    <processing requestID="router1-np">
        <function>Classify</function>
        <function>Modify</function>
    </processing>
    <!-- We want buffer space to queue outgoing packets from the NP -->
    <routing:storage requestID="router1-oqueue">
        <capacity units="KB">
            <value>50</value>
        </capacity>
        <persistence>transient</persistence>
        <!-- Specify that this storage is the output queue for the NP -->
        <routing:queue>
            <routing:direction>outgoing</routing:direction>
            <routing:queueFor>router1-np</routing:queueFor>
        </routing:queue>
    </routing:storage>
    <!-- Request a specific interface on the router, which we will get
         all to ourselves -->
    <communication reqestID="router1-interface1"
        physicalID="geni.backbone.pop.seattle.prouter1.interface3">
        <capacity units="Gbps">
            <value>10</value>
        </capacity>
        <medium>exclusive</medium>
    </communication>
</component>
<component type="virtual router" requestID="router2"
        physicalID="geni.backbone.pop.sanjose.prouter3">
... Contents look very similar to router1 above
</component>
<!-- Now we request to use a lambda on a specific fiber -->
<link type="point-to-point" requestID="backbone"
    physicalID="geni.backbone.fiber.seattle-sanjose">
    <endpoint requestID="router1-lambda"
        physicalID="geni.backbone.pop.seattle.prouter1.interface3 ">
    </endpoint>
    <endpoint requestID="router2-lambda"
        physicalID="geni.backbone.pop.sanjose.prouter3.interface1 ">
    </endpoint>
    <!-- Here, FreqShared indicates that the fiber is shared by several
         different labmdas (wavelengths) -->
    <medium>FreqShared</medium>
    <mediumtype>PointToPoint</mediumtype>
    <!-- It is quite likely that '10Gbps' is not the correct way to specify
         the capacity of a lambda. Thus, this will likely be replaced by
         some more-appropriate metric in an 'optical' name space -->
    <capacity units="Gbps">
        <value>10</value>
    </capacity>
</link>
```

To be written. In this section, we put RSpec in context, by giving examples of RSpecs for components important to major GENI communities such as the wireless community and the router/backbone community.

7.2 Certificates

Certificates are used to identify principals that need to sign data objects and messages within the GENI system.

> **Engineering Note:** Although further elaboration of the above is required, the GENI *architecture* places fairly minimal requirements on certificates. The establishment of appropriate models and procedures for a GENI certificate authority is required, but is an operational issue.
>
> **Technology Decision:** We propose that the certificate format be a simple XML document as defined by the XML-SEC standard. An alternative is to generalize the architecture to support multiple certificate formats rather than require XML-SEC. For example, the x509 standard for digital certificates provides roughly the same capabilities and could be used as an alternative.

7.3 Tickets

A component owner signs an RSpec—one that includes the right to allocate the corresponding resources—to produce a ticket. Such tickets are either returned by GetTicket or delegated by GrantTicket, and they can be "redeemed" at a later time using the CreateSlice, ModifySlice, or SplitSlice operation. The GMC needs to define the format for tickets, and the set of rules governing their delegation.

A ticket includes at least the following information:

1. An RSpec, describing the resources for which rights are being granted by the component owner.
2. A Slice Name, indicating the slice, or slice authority, to which rights to allocate the resources are being granted
3. A timestamp indicating for how long the ticket is valid. The ticket must be "redeemed" before this time (with a CreateSlice() operation).
4. A timestamp indicating the period of time during which the resources specified in the ticket are available to be consumed by the slice.
5. An indication of specific rights or privileges associated with the ticket.

> **Unresolved Issue:** It is not clear if this problem gets solved at the certificate level, the Ticket level, or the RSpec level, but we need to ensure that a ticket uniquely identifiable so that the resources it denotes cannot be applied multiple times. This is an accountability issue, and the requirement is that all tickets be auditable, so that the creator or chain of creators of the ticket can be reconstructed. (NB: most likely this issue is addressed at the Ticket level, since RSpecs are used elsewhere in the system where this is not an issue, and Certificates are simply authoritative identifiers.)
>
> **Unresolved Issue:** Should the timestamps (particularly the resource availability one) be time intervals instead? Also, since intervals are associated with the ticket

rather than the RSpec, the GetTicket() operation appears to require the desired intervals as an additional argument.

Unresolved Issue: What specific rights and privileges must be representable within a ticket? (The present text appears not to require any, since all tickets are used for the same, single purpose.)

8 Functional Descriptions

The discussion to this point has focused on GMC abstractions and interfaces, remaining relatively silent as to how administrators manage components and researchers control slices (experiments). This section sketches how we expect these key functions will be implemented, primarily through the use of aggregates.

It is important to keep in mind that this section outlines several possible scenarios for how the GENI management plane organizes itself into a collection of aggregates to accomplish key functions. Other structures are possible, the only requirement being that they adhere to the set of abstractions and interfaces defined in the previous sections.

8.1 Resource Allocation

Allocation of physical substrate resources to user experiments (in the form of slices) is the central function of the GMC. This section describes the basic resource allocation model used by components, discusses ways the model may be expanded to larger granularity allocations, and describes some policy considerations.

8.1.1 Basic Model

Resource allocation in GENI is the process by which a potential user of a component's resources decides, and then obtains the right, to do so.

This process proceeds as follows.

1. A component (or more precisely, its CM) advertises the availability of its resources by registering itself, and advertising its resources using an RSpec description in "advertising" fashion. The registration process makes the component's advertisement of its capabilities available to others. Section 8.3 describes this process in more detail.
2. A user or higher-level embedding service, acting on behalf of a slice, determines its potential interest in the component by examining the component's advertised capabilities and attributes.
3. Optionally, a potentially interested user or higher-level embedding service queries the component directly for further information about its capabilities and attributes to make a final determination of interest.
4. The user or higher-level embedding service, having determined that it wishes to acquire resources on the component, executes a GetTicket operation, specifying an RSpec in "request" fashion to request the desired capabilities.

5. The component responds with a Ticket granting the request (or, returns a failure). This gives the user's slice rights to actually allocate the resources at an appropriate time.
6. At the desired time, the user or embedding service invokes a CreateSlice operation on the component, to actually allocate the component's resources to the slice.

> **Unresolved Issue:** The component interface used to support the optional query step in the two-pass resource discovery protocol has not been defined.

8.1.2 Coordination Aggregates

It is often the case that slivers are created in a coordinated fashion across an aggregate of components. For example, a backbone aggregate might coordinate link bandwidth allocation across a set of programmable core router components. Similarly, a wireless subnet aggregate might oversee spectrum allocation among a set of access-point components. In both examples, the aggregate—which we generically call a *coordination aggregate*—effectively provides a common interface for a related set of components, treating the set as a unit for the purpose of allocating resources, instantiating slices, and starting and stopping slices. We sometimes say that a coordination aggregate implements a *slice control* function.

It is likely that coordination aggregates export exactly the same interface as the underlying components. For example, to create a slice that spans a backbone, a user might invoke CreateSlice on the backbone aggregate, which in turn calls CreateSlice on the individual backbone components. In this case, the aggregate maintains state for the slice—the set of components it runs on and the quantity of resources have been bound to it. As another example, one could imagine a root aggregate that represents all of GENI's components; the root would be an aggregate of aggregates.

> **Unresolved Issue:** It is important to recognize the implication of this approach, which is that in the end a single RSpec must be able to advertise the entire potential resource configuration of GENI. It is possible that alternate approaches will be needed to maintain scalability and enhance separation of concerns.

A hierarchy of aggregates is formed through the invocation of GetTicket and GrantTicket operations. For example, a set of programmable core nodes might explicitly grant control over 100% of their resources to a backbone aggregate. Alternatively, an aggregate might be created implicitly by sharing a code base with a set of constituent components; such an aggregate would simply to programmed (hardwired) to "know" what resources are available on the set of components.

8.1.3 Researcher Portals

A *portal* is an interface—graphical, programmatic, or both—that defines an "entry point" through which users access GENI components. A portal is likely implemented by a combination of services. For example, a portal might export a user-friendly slice embedding interface, which is implemented by composing a *resource discovery service* that learns about a set of available components, a *resource allocation service* that requests (negotiates) resource acquisition, and a *slice stitching service* that creates a set of slivers that span one or more component aggregates. Beyond slice creation, a portal might also provide an interface to experiment configuration and

management services, as well as other user-level services that contribute to the overall researcher experience.

Different user communities can define portals tailored to the needs of that community, with each portal defining a different model for slice behavior, or support a different experimental modality. For example, one portal might create and schedule slices on behalf of researchers running short-term controlled experiments, while another might acquire resources needed by slices running long-term services.

> **Engineering Note:** In general, we expect portals to take advantage of web services. For example, sub-services might register with higher level services, and so on recursively, with high-level, user-friendly services ultimately exporting web-based graphical interfaces. Once such a nesting of service registrations are available, others can build on them in ways that do not adhere to a simple hierarchy, potentially resulting in multiple portals.

8.1.4 Resource Allocation Policies

Resource allocation is fundamentally decentralized in GENI with each component owner defining the policy by which it is willing to hand out resources in response to GetTicket calls. We envision such a policy being represented (internal to the component) as a set of

> slice name : RSpec

bindings. Components might allocate resources directly to individual slices, to an network-specific aggregate that manages resource allocation across a set of related components, to a researcher portal that implements a more sophisticated resource allocation policy, or to some combination of all three.

For convenience, control over resources might be concentrated (centralized) in a single portal that implements an allocation policy on behalf of the GENI Science Council.[3] Alternatively, the Science Council might mandate that resources be split (in some ratio) between a portal that distributes resources to short-term controlled experiments and a portal that distributes resources to slices running long-term services. The first portal might allocate resource by enforcing FCFS access to a set of components, while the other might implement a market-based "brokerage" service.

We conclude this discussion of resource allocation with two observations. First, the proceeding discussion implies that components make arbitrarily fine allocations of every last ounce of their resources, but this is not necessarily the case. For example, a component could elect to hand out only a fraction of its total capacity in this way (50% to one brokerage service and 20% to

---

[3] In effect, the Science Council is the component owner with respect to allocation policy, independent of where the components are physically located. While this may be true for NSF-funded components, it will not be true for the larger GENI ecosystem, in which many parties are expected to connect resources to GENI. We also need to revisit the issue of how "ownership responsibility" is shared between the Science Council and hosting sites.

another), but then hand out tickets that grant the caller the right to a "fair share" of the remaining capacity. This fair share is likely what is allocated (by default) to a slice upon creation.

Second, the architecture is neutral about the possibility of overbooking resources. It is likely (and perhaps should be required) that individual components not hand out tickets that they later have to reject, but aggregates could manufacture tickets that are only speculatively bound to resources. Note that an aggregate can return a ticket that is directly redeemed on a component, or it can return a ticket that can only be redeemed on the aggregate. Only the latter could be used to support overbooking if individual components are not allowed to overbook.

8.2 Facility Management

While individual owners could be responsible for managing (providing operational support for) their components, we envision this role being played on behalf of owners by one or more *management authorities* (MA), each of which has the following responsibilities for what is generally regarded as *operations and maintenance (O&M) control*. O&M control includes:

- booting components into the right state,
- detecting, debugging, and recovering failed components,
- responding to user queries (diagnosing user-perceived problems),
- responding to security complaints from aggrieved parties, and
- monitoring the system for AUP compliance.

The way in which nodes securely boot in PlanetLab illustrates one possible implementation of O&M control for GENI components [GDD-06-39].

A management authority likely provides an operations center—staffed by a set of administrators—but the MA also manifests itself as an aggregate that exports an interface that can be invoked by other objects in the GENI system. For example, a backbone aggregate might provide O&M control for all the components that make up a backbone network, and another aggregate might do the same for the components distributed across a particular wireless subnet.

We expect such aggregates to support the standard aggregate interface (e.g., ListComponents returns a reference to all components managed by this MA), extended with aggregate-wide variants of the component-level informational operations defined in Section 4.5: ListSlices, GetStatus, GetResponsibleSlice. These operations serve as underpinnings for higher level management services provided by the Management Authority.

> **Engineering Note:** While each aggregate exports a public interface to the rest of GENI—e.g., ListComponents, GetResponsibleSlice—each MA is free to establish a private interface that it uses to manage its components. Although a private interface, we expect any MA that is part of GENI proper to make this interface open (i.e., to publish it); MAs in the larger GENI ecosystem are not required to do so. Note that this private interface need not be XML-RPC based. Examples of

O&M control that might be used in GENI can be found in companion documents [GDD-06-13, GDD-06-15, GDD-06-25].

8.3 Registering Components

This section describes how a component is assigned a GGID and registers itself in the global component registry, working in concert with the management authority (MA) that is responsible for the component's correct operation. The process assumes a global GENI component repository, where both this repository and the MA already have their own GGIDs.

The process results in the component being assigned a GGID and ready to authenticate itself and respond to requests sent to its URI. In addition, the registry will have allocated a name to the component, and will have recorded information about the component's resources and access constraints. This information can be used during resource discovery, corresponding to step (1) in Section 8.1.1.

> **Engineering Note:** This discussion assumes the component manager runs on the component itself, and so the two are trivially bound together. In general, a component's manager may run on another machine, effectively serving as a proxy for the component [GDD-07-42]. In this case, additional work is required to bind a component to its manager.

As an example, the first step might be to assign a GGID to the component, signed by the MA responsible for the component. The component communicates with the MA according to whatever O&M control protocol the MA supports (see Section 8.2), at which time the MA generates a name (e.g., geni.us.backbone.nyc), a UUID, and a key pair for the component, signs the component's UUID and public key to form a GGID, and passes the name, the new key pair, the component's GGID, and the MA's GGID to the component. In return, the component informs the MA of the URI at which it will process requests.

At this point, the MA passes the following additional information to the component: (1) URIs of authorities and public keys necessary to validate GGIDs,[4] and URIs and GGIDs of registries in which this component is to register. The MA and component must also establish a shared definition of what resources are available on the component. This involves the component reporting various physical limits (e.g., available memory or bandwidth), the MA normalizing this information according to any aggregate-wide policies and conventions, and the MA representing this information as an Rspec.

Finally, the component contacts the registries indicated by the MA, confirms that it has established communication with the GGID it was given by the MA, and registers the name, GGID, and RSpec for itself. The registry verifies that the name is legitimate (i.e., can be assigned

[4] The root of the GGID authority needs to be included, but other keys may be passed to simplify validations of aggregates or local services. Components trust their MAs to configure them to validate GGIDs in this step. Hereafter all GGIDs are validated and URIs authenticated.

by the component's MA), and assuming the registration completes successfully, the component becomes visible to all of GENI.

9 Example Deployment

We conclude by sketching a possible deployment of GENI proper in terms of the GMC framework defined up to this point. The sketch includes an example set of components, along with a set of aggregates and infrastructure services that define how the components are accessed and managed.

9.1 Component Substrate

The GENI architecture is neutral on the set of components that populate the substrate—and we expect the set of physical devices to change over time—but we can identify an example set of components that will likely define GENI, at least initially. The following primarily serves to establish a common set of terminology for the GENI substrate. A companion set of documents will give a reference design for each component type [GDD-06-13, GDD-06-19, GDD-06-26, GDD-06-21, GDD-06-22].

GENI proper will include a US-wide backbone network spanning on the order of 25 Points-of-Presence (PoPs). The PoPs will be connected by an underlying *fiber facility* with a *programmable core node* (PCN) component available at each PoP. These PCNs will include a combination of general-purpose processors and specialized blades (e.g., FPGAs and network processors) connected by a high-speed backplane.

GENI proper will include a set of 100-200 edge sites, each of which will host a *programmable edge cluster* (PEC) component. These PECs will range over a wide spectrum, from two or three processors connected by a low-end commodity switch, to a large cluster potentially including hundreds of processors and a high-performance interconnect. Each PEC (and hosting site) will be connected to the backbone by a *tail circuit* to a backbone PoP. These tail circuits will exploit a wide range of technologies, including dedicated fiber, leased layer-2 circuits, and tunnels through the commodity Internet. Each PEC will also be connected to the commodity Internet via whatever connectivity is provided by the hosting site.

GENI proper will include five wireless subnets, each built around a different wireless technology. Each subnet will include (1) a *programmable edge node* (PEN) component that connects the subnet to both a near-by GENI PoP (via a tail circuit) and to the commodity Internet (via a site-specific connection); (2) a set of *programmable wireless nodes* (PWN) components that include wireless interface cards; and (3) an access network that connects the PEN and the set of PWNs.

Figure 9.1 depicts the various components configured in a hypothetical "Site A." In practice, a given site is not likely to include multiple wireless subnets: most sites will include either a single wireless subnet or just an edge cluster. Moreover, sites that do include multiple technologies will likely consolidate the PEC and PEN into a single component.

> **Engineering Note:** Figure 9.1 also includes an *GENI Gateway* (GGW), which is a device that connects the PEC/PEN to both the GENI tail circuit and to the

hosting site's existing network. This GGW is not programmable (and not a stand-alone component of GENI), but rather, it is managed as part of the adjacent PEC / PEN. The exact specification for a GGW is yet to be determined, and likely depends on the precise scenario in which it is deployed. When the tail circuit is at layer-2 or lower, the GGW clearly must not be limited to IP. On the other hand, the GGW must be able to route to the commodity Internet. It is also not clear whether the GGW must provide QoS support, or if this functionality is subsumed by the PEN / PEC. In addition to providing connectivity, the GGW might also implement some or all of the component containment functionality (see the Engineering Note at the end of Section 3.1.3). Assuming the GGW sits at a choke point at the site, such functionality could serve all PEC and PEN components at that site.

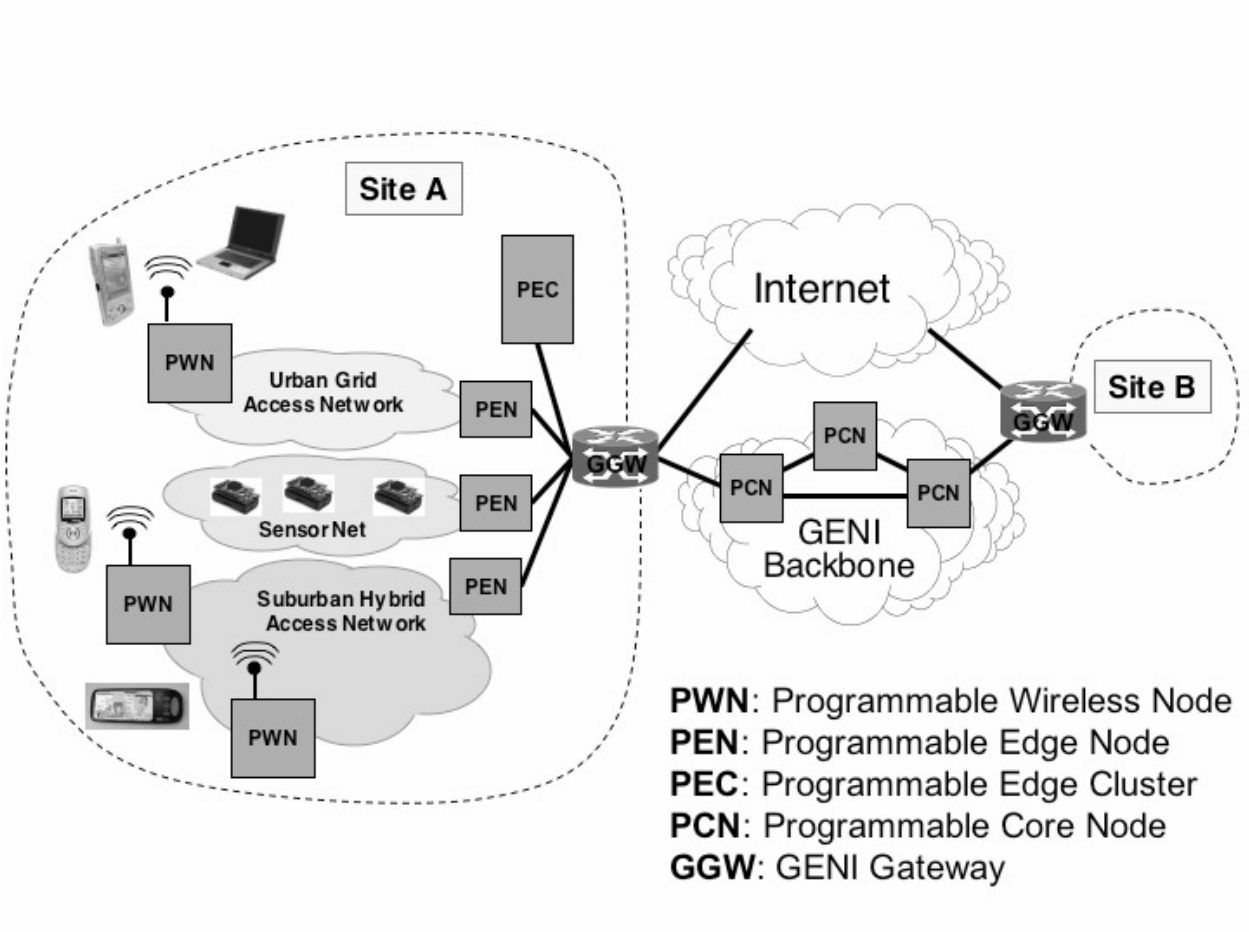

Figure 9.1: Example GENI Substrate

9.2 Management Structure

Figure 9.2 illustrates a management structure that corresponds to the example substrate shown in Figure 9.1. The example is limited to GENI proper, and assumes that each component runs a component manager (not shown).

First, a set of *management authorities* is responsible for collections of similar components connected by a common network technology. Here, we assume a distinct *management aggregate* is associated with each management authority, and that this aggregate implements both the O&M and slice control functions described in Sections 8.1 and 8.2, respectively. These aggregates may be co-located with the managed set of components, but this is not necessarily the case (and is not possible in the case of the backbone and edge site aggregates, which correspond to a widely distributed set of components). Note that the exact way in which a management aggregate boots and controls a set of components is defined in elsewhere, in a component-specific way [GDD-06-13, GDD-06-15, GDD-06-25, GDD-06-39].

Second, a pair of portals provides interfaces to all of GENI proper. Researchers request slices and control their experiments using the *researcher portal*, where all rights for resources are concentrated with this portal and the Science Council policy is implemented by this portal. The portal then contacts the slice control interface in an appropriate set of management aggregates to create the corresponding slivers. Similarly, an *operator portal* defines a GENI-wide interface for operations across all GENI components. This portal, in turn, contacts the O&M control interface for the appropriate set of subnet-specific management aggregates.

Although not explicitly shown, both portals serve as front-ends for a set of infrastructure services that researchers engage to help them manage their slices, and operators engage to help them monitor and diagnose the components [GDD-06-24]. Many of these services actually run in a slice on the constituent components.

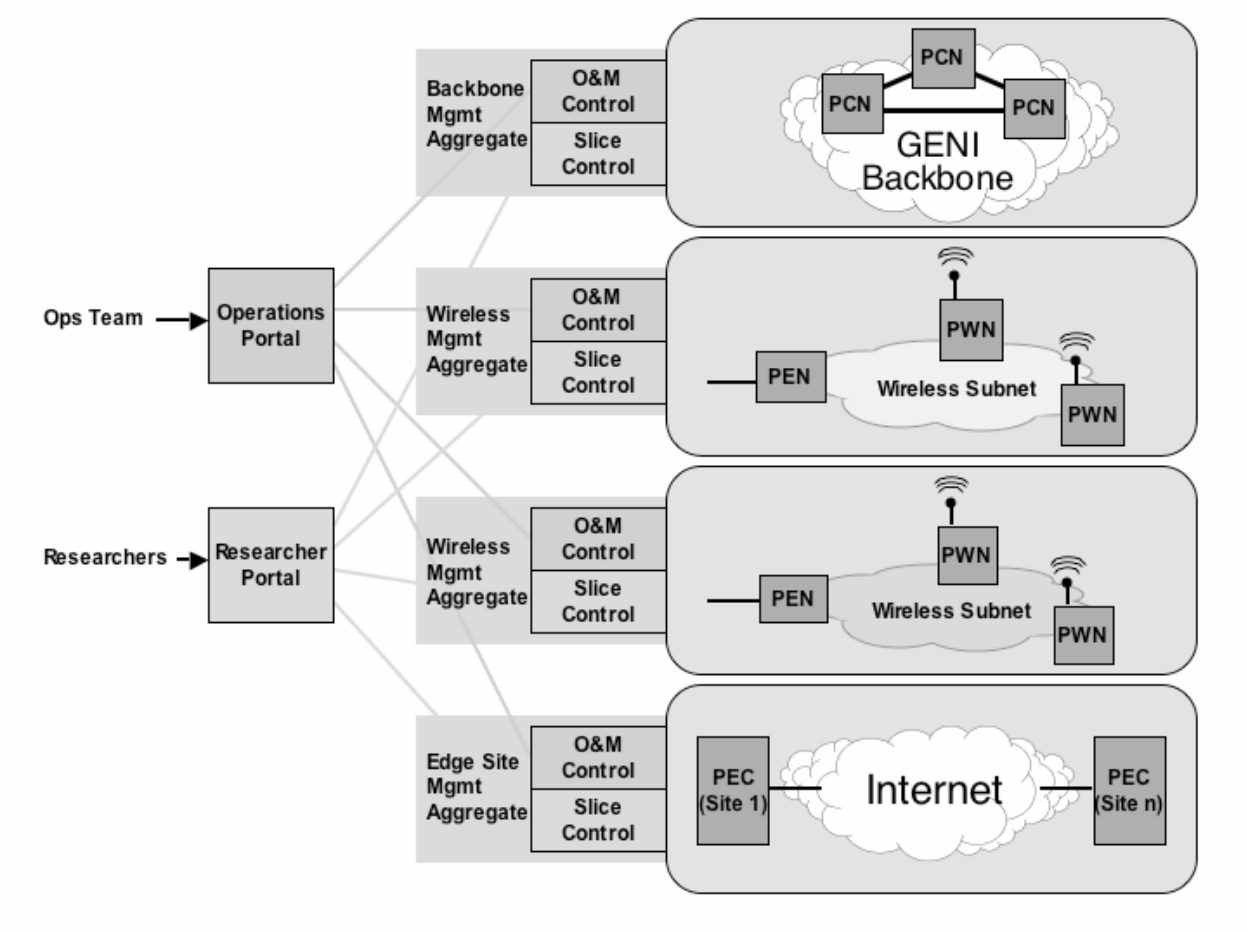

Figure 9.2: GENI Management Structure

References

[GDD-06-07] Larry Peterson (Ed), "GENI: Conceptual Design, Project Execution Plan," *GENI Design Document 06-07*, GENI Planning Group, January 2006.

[GDD-06-12] Paul Barford (Ed), "GENI Instrumentation and Measurement Systems (GIMS) Specification," *GENI Design Document 06-12*, Facility Architecture Working Group, September 2006.

[GDD-06-13] Jack Brassil (Ed), "GENI Component: Reference Design," *GENI Design Document 06-13*, Facility Architecture Working Group, September 2006.

[GDD-06-15] Dipankar Raychaudhuri, Sanjoy Paul, "Requirements Document for Management and Control of GENI Wireless Networks," *GENI Design Document 06-15*, Wireless Working Group, September 2006.

[GDD-06-21] Chip Elliott, "System Engineering Document for GENI Urban Wireless Grid," *GENI Design Document 06-21*, Wireless Working Group, September 2006.

[GDD-06-22] Chip Elliott, "System Engineering Document for GENI Suburban Wireless Network," *GENI Design Document 06-22*, Wireless Working Group, September 2006.

[GDD-06-23] Thomas Anderson and Michael Reiter, "GENI Facility Security," *GENI Design Document 06-23*, September 2006.

[GDD-06-24] Thomas Anderson (Ed), "GENI Distributed Services," *GENI Design Document 06-24*, Distributed Services Working Group, September 2006.

[GDD-06-25] T.V. Lakshamn, Sampath Rangarajan, Jennifer Rexford, Hui Zhang, "Backbone Software Architecture," *GENI Design Document 06-25*, Backbone Working Group, September 2006.

[GDD-06-26] Dan Blumenthal and Nick McKeown, "Optical Node Architecture," *GENI Design Document 06-26*, Backbone Working Group, September 2006.

[GDD-06-39] Larry Peterson, Aaron Klingaman, Steve Muir, Mark Huang, Andy Bavier, Vivek Pai, and KyoungSoo Park. "Management Authority: Reference Implementation" *GENI Design Document 06-39*, December 2006.

[GDD-07-42] John Wroclawski. "Using the Component and Aggregate Abstractions in the GENI Architecture" *GENI Design Document 07-42*, January 2007.

[GMC Spec] Ted Faber (Ed), "GMC Specifications," http://www.geni.net/wsdl.php, Facility Architecture Working Group, September 2006.